1. Algoritmos de Ordenação

1.1. Insertion Sort

O algoritmo de ordenação por Insertion sort é indicado em situações nas quais os dados chegam ao **longo do tempo**, dado que a sua premissa consiste em inserir cada elemento em sua posição correta no vetor de dados. A Figura 1 apresenta um exemplo de funcionamento do algoritmo Insertion sort, em que o elemento que está sendo comparado na iteração atual (variável v no algoritmo) encontra-se destacado em amarelo.

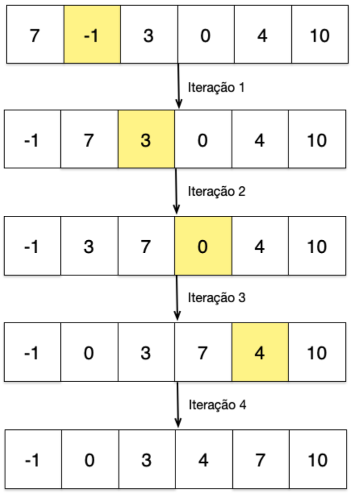

Figura 1: Funcionamento da técnica Insertion Sort.

O algoritmo abaixo implementa a técnica Insertion sort, cuja análise de complexidade é dada da seguinte maneira:

- Complexidade de espaço: $\theta(n)$
- Complexidade de tempo: $O(n^2)$
- Estável: Sim
- In-place: Sim

In [1]:
```
from matplotlib import pyplot
import numpy
import math
import timeit
import queue
import sys
```

In [2]:
```
def InsertionSort(A):
    n = len(A)
    for i in range(2,n):
        v = A[i]
        j = i-1

        while (A[j] > v) and (j >= 0):
            A[j+1] = A[j]
            j = j-1
        A[j+1] = v
```

Note que o laço mais **externo** executa $n - 1$ vezes, ou seja, $\theta(n)$ vezes. A complexidade do algoritmo de Insertion sort é dominada pelo laço mais **interno**. Caso tenhamos um vetor ordenado de forma **crescente**, por exemplo, a condição `A[j+1] > v` nunca será satisfeita e, por conseguinte, o laço mais interno não será executado. Nesta situação hipotética, a complexidade seria então dada por $\theta(n)$, que seria então a situação de **melhor caso** para o Insertion sort.

Entretanto, no **pior caso**, ou seja, um vetor ordenado de forma **decrescente**, o laço mais interno será executado $i - 1$ vezes para cada valor de i. Desta forma, teremos $i(i-1) \approx i^2$ execuções, resultando em uma complexidade final de $O(n^2)$.

Exemplo de funcionamento:

In [3]:
```
A = numpy.random.randint(-20, 20, 10)
print('Vetor de entrada: '+ str(A))
InsertionSort(A)
print('Vetor ordenado: '+ str(A))
```

```
Vetor de entrada: [ -3  -7  15 -12 -18 -16 -16  10   2  11]
Vetor ordenado: [-18 -16 -16 -12  -3  -7   2  10  11  15]
```

1.2. Merge sort

O algoritmo de ordenação por Merge sort é baseado na metodologia de **divisão e conquista**, em que o problema principal é dividido em subproblemas que são resolvidos e, posteriormente, agrupados com o intuito de gerar a solução final. A divisão é realizada em $\theta(1)$, ao passo que a conquista é realizada em $\theta(n)$. De maneira geral, a ténica realiza divisões no vetor pela metade, até que um único elemento é obtido. Em seguida, o processo de conquista é realizado, em que os elementos são agrupados de maneira ordenada. A Figura 2 apresenta esse processo, em que os elementos cinza denotam a primeira metade do vetor, e os elementos em azul correspondem ao final da etapa de divisão.

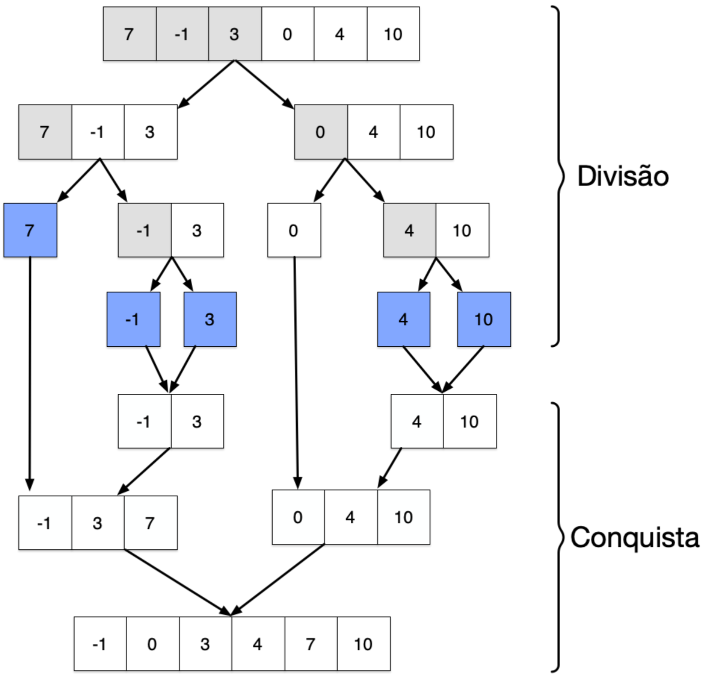

Figura 2: Funcionamento da técnica Merge Sort.

O algoritmo abaixo implementa a técnica Merge sort, cuja análise de complexidade é dada da seguinte maneira:

- Complexidade de espaço: $2\theta(n)$
- Complexidade de tempo: $\theta(n \log n)$
- Estável: Sim
- In-place: Não

A complexidade de tempo pode ser calculada por meio do **Teorema Mestre** da seguinte maneira:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{caso contrário.} \end{cases} \tag{1}$$

Como pode ser observado, a função `MergeSort` é invocada duas vezes de maneira recursiva, em que cada chamada é realizada em metade do vetor. Desta forma, temos que $a = b = 2$.

Entretanto, falta ainda analisarmos o termo **residual** $f(n)$ da Equação 1, o qual é composto por três laços: (i) o primeiro deles copia a primeira metade do vetor de dados de entrada A para o vetor **auxiliar** B e possui complexidade de tempo $\theta\left(\frac{n}{n}\right) \in \theta(n)$; (ii) o segundo laço copia a segunda metade do vetor A em B, porém de maneira invertida, e também possui complexidade de tempo $\theta\left(\frac{n}{n}\right) \in \theta(n)$; e (iii) o último laço copia os elementos de B no vetor de dados de entrada A de maneira ordenada, ou seja, nesta etapa é realizada a junção (*merge*) dos dados, e possui complexidade $\theta(n)$. Desta forma, o termo residual, ou seja, a conquista, possui complexidade $\theta(n)$.

Assim sendo, podemos reescrever a Equação 1 da seguinte maneira:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{caso contrário.} \end{cases} \tag{2}$$

O termo residual pode ser mapeado para a forma cn^k assumindo $c = k = 1$. Desta forma, de acordo com o Teorema Mestre, a complexidade final de tempo do Merge Sort é dada por $\theta(n^k \log_b n) = \theta(n \log n)$.

In [4]:
```
def MergeSort(A, e, d):
    if (d > e):

        #Divisão é realizada em theta(1) *****
        m = math.floor((e+d)/2)
        MergeSort(A, e, m)
        MergeSort(A, m+1, d)
        # Fim da divisão ********************

        # Conquista é realizada em theta(n) ***
        B = numpy.zeros(d+1);

        for k in range(0, m+1):
            B[k] = A[k]

        j = d
        for k in range(m+1, d+1):
            B[j] = A[k]
            j = j-1

        i = e
        j = d
        for k in range(e, d+1):
            if(B[i] < B[j]):
                A[k] = B[i]
                i = i+1
            else:
                A[k] = B[j]
                j = j-1
        # Fim da conquista *******************
```

Exemplo de funcionamento:

In [5]:
```
A = numpy.random.randint(-20, 20, 10)
print('Vetor de entrada: '+ str(A))
MergeSort(A, 0, 9)
print('Vetor ordenado: '+ str(A))
```

```
Vetor de entrada: [ -3  -1  15  -3 -17   7  -9  19 -18   6]
Vetor ordenado: [-18 -17  -9  -3  -3  -1   6   7  15  19]
```

1.3. Quicksort

O algoritmo de ordenação Quicksort também é baseado na metodologia de divisão e conquista, em que esta última é realizada em tempo $\theta(1)$, ao passo que o "gargalo" da técnica está justamente na etapa de divisão. O algoritmo faz uso de um elemento denominado **pivô**, e assume que os elementos à **esquerda** deste são menores ou iguais à ele; ao passo que os elementos à sua **direita** são maiores ou iguais ao pivô. Note que, não necessariamente, os elementos que estão à esquerda e à direita estarão ordenados entre si.

Desta forma, a complexidade de tempo do Quicksort depende, necessariamente, da **posição** do pivô durate a execução do algoritmo. Se conseguíssemos garantir que o pivô sempre dividisse o vetor de entrada ao meio, então a a complexidade de ordenação seria igual à do Merge sort, ou seja, $\theta(n \log n)$. Entretanto, essa garantia não é possível de ser assegurada. A Figura 3 apresenta o funcionamento da técnica Quicksort, em que os elementos à esquerda do pivô são representador pela cor cinza, os elementos azuis correspondem aos valores que são maiores ou iguais ao pivô e, finalmente, o pivô está representado em laranja.

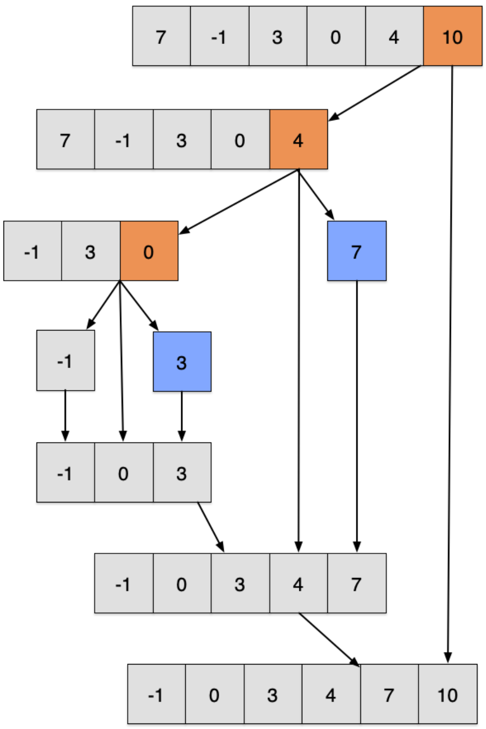

Figura 3: Funcionamento da técnica Quicksort.

O algoritmo abaixo implementa a técnica Quicksort, cuja análise de complexidade é dada da seguinte maneira:

- Complexidade de espaço: $\theta(n)$
- Complexidade de tempo: $O(n^2)$, porém sua complexidade no caso médio é de $O(n \log n)$
- Estável: Não
- In-place: Sim

In [6]:
```
def Quicksort(A, e, d):
    if (d > e):
            index = Partition(A, e, d) #função que retorna o novo í
ndice do pivô (elemento laranja)
            Quicksort(A, e, index-1) # executa o Quicksort na parte
cinza
            Quicksort(A, index+1, d) # executa o Quicksort na parte
laranja

def Partition(A, e, d):
    pivot = A[d]
    p_index = e

    for i in range(e,d):
        if (A[i] <= pivot):
            A[i], A[p_index] = A[p_index], A[i]
            p_index = p_index + 1

    A[p_index], A[d] = A[d], A[p_index]

    return p_index
```

Exemplo de funcionamento:

In [7]:
```
A = numpy.random.randint(-20, 20, 6)
print('Vetor de entrada: '+ str(A))
Quicksort(A, 0, 5)
print('Vetor ordenado: '+ str(A))
```

```
Vetor de entrada: [ -9  10 -11   5 -16 -20]
Vetor ordenado: [-20 -16 -11  -9   5  10]
```

1.4. Heapsort

O algoritmo de ordenação Heapsort é uma técnica baseada em comparações, cujo funcionamento é dividido em três partes: (i) construção de um heap (usualmente binário) a partir dos dados de entrada, (ii) remover a raiz e inserir na última posição do vetor, e (iii) mover último nó para a posição da raiz. Note que este último passo exige que o heap seja atualizado a cada remoção de tal forma a manter suas propriedades. Para uma ordenação **crescente**, devemos utilizar um **heap máximo**, ao passo que para uma ordenação **decrescente** deve-se empregar um **heap mínimo**.

A primeira parte, como citado anteriormente, consiste em transformar um vetor de entrada em um **heap binário**, o qual, por sua vez, utiliza uma estrutura de **árvore binária**. Isto pode ser feito, basicamente, por meio de uma associação de índices do próprio vetor, como apresenta a Figura 4.

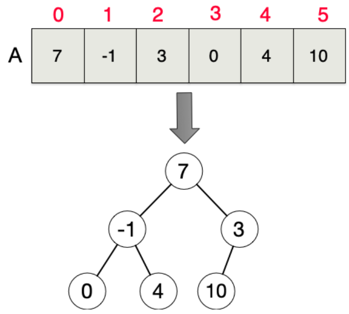

Figura 4: Mapeando um vetor para uma árvore binária.

Neste caso, temos que a raiz é representada pelo elemento $A[0]$, cujos filhos são os elementos $A[1]$ e $A[2]$. Por conseguinte, os filhos do nó $A[1]$ são dados por $A[3]$ e $A[4]$, e o último nó representado por $A[5]$. Assim sendo, temos que:

- A[0]: possui como filhos $A[1]$ e $A[2]$, ou seja, $A[i+1]$ e $A[i+2]$ para $i=0$.
- A[1]: possui como filhos $A[3]$ e $A[4]$, ou seja, $A[i+2]$ e $A[i+3]$ para $i=1$.
- A[2]: possui como filho $A[5]$, ou seja, $A[i+3]$ para $i=2$.

Podemos, então, generalizar da seguinte forma: $A[i]$ possui como filhos os elementos $A[2i+1]$ e $A[2i+2]$.

Em seguida, precisamos verificar se o heap obedece à propriedade de um heap máximo, ou seja, o valor armazenado em um nó pai tem que ser maior ou igual ao(s) valor(es) armazenzado(s) em seu(s) nó(s) filho(s). A Figura 5 apresenta essa metodologia.

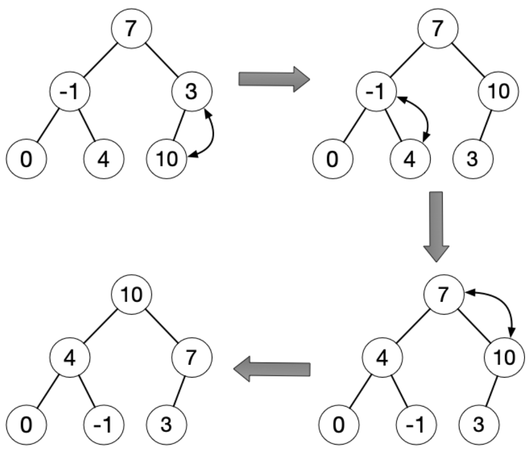

Figura 5: Mapeando uma árvore árvore binária para um heap máximo.

Em seguida, o nó raiz (maior valor) é removido e inserido na última posição do vetor, e o último nó do heap toma o seu lugar. Entretanto, após esta operação, precisamos verificar se o heap obedece à propriedade de heap máximo, ou seja, um nó filho tem sempre um valor menor ou igual ao seu pai (procedimento implementado pela função `Heapify` no algoritmo abaixo). Após atualizarmos a estrutura, o procedimento todo é repetido novamente até que todos os elementos tenham sido removidos do heap. A Figura 6 ilustra este procedimento.

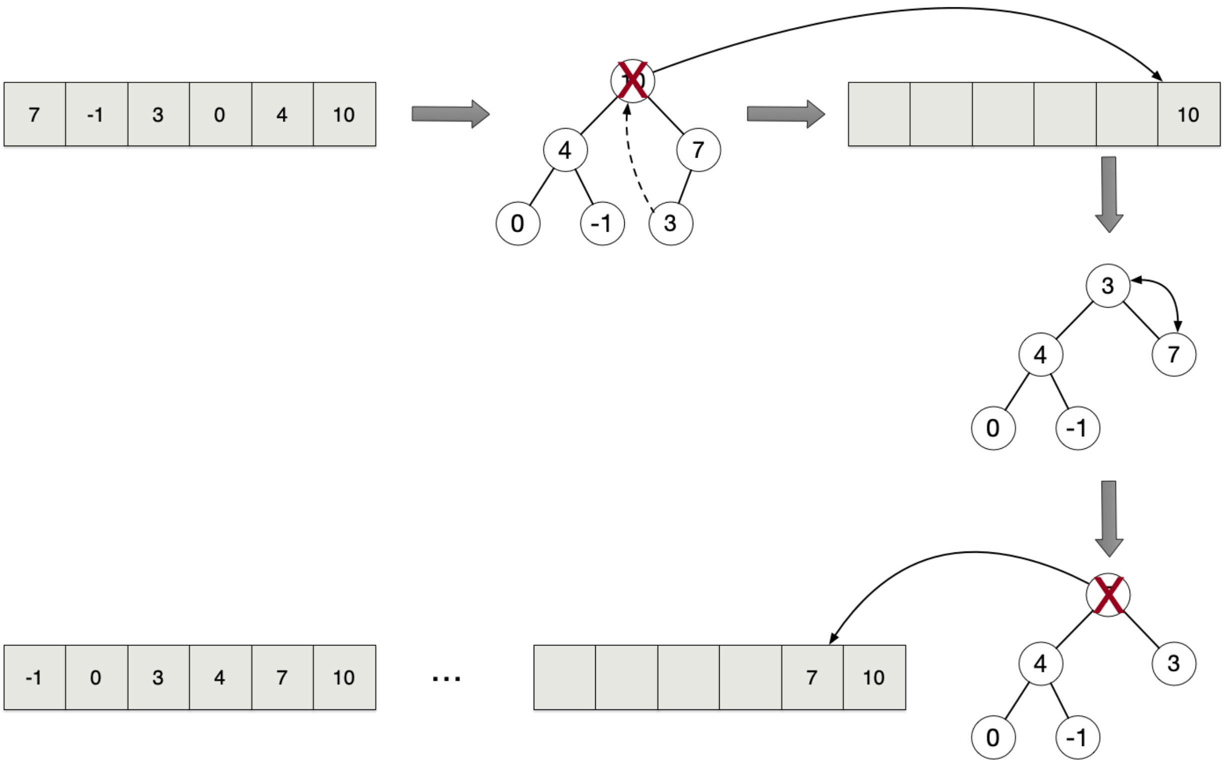

Figura 6: Funcionamento da técnica Heapsort.

O algoritmo abaixo implementa a técnica Heapsort, cuja análise de complexidade é dada da seguinte maneira:

- Complexidade de espaço: $\theta(n)$
- Complexidade de tempo: $O(n \log n)$
- Estável: Não
- In-place: Sim

In [8]:

```
#Algoritmo baseado no código disponível em https://www.tutorialspoi
nt.com/python-program-for-heap-sort

# função que verifica se um heap obedece às propriedades de heap má
ximo
def Heapify(A, n, i):

    maior = i # maior valor
    l = 2*i + 1 # filho à esquerda
    r = 2*i + 2 # filho à esquerda

    # verifica se filho à esquerda existe e quem possui o maior val
or
    if (l < n) and (A[i] < A[l]):
        maior = l

    # verifica se filho à direita existe e quem possui o maior valo
r
    if (r < n) and (A[maior] < A[r]):
        maior = r

    # analisa raiz
    if (maior != i):
        A[i],A[maior] = A[maior],A[i] # faz a troca entre a raiz e
o maior valor encontrado

        # executa todo o processo novamente
        Heapify(A, n, maior)

# Heapsort
def Heapsort(A):

    n = len(A)

    # cria o heap máximo
    for i in range(n-1, -1, -1):
        Heapify(A, n, i)

    # remoção da raiz e sua inserção na posição correta do vetor or
denado
    for i in range(n-1, 0, -1):

        A[i], A[0] = A[0], A[i]
        Heapify(A, i, 0)
```

Note que o primeiro laço da função `Heapsort` executa $n + 1$ vezes, ou seja, $\theta(n)$ vezes, ao passo que o segundo laço executa, também, $\theta(n)$ vezes. Agora, precisamos calcular a complexidade da função `Heapify` , a qual objetiva manter a estrutura de acordo com as propriedades de um heap máximo, conforme mencionado anteriormente. Como a criação do heap objetiva a construção de uma árvore binária balanceada, sabemos que sua altura pode ser dada por $O(\log n)$. No pior caso, ou seja, quando um nó folha precisa ser movido até a raiz, temos que um número $O(\log n)$ de comparações deve ser efetuado até que a propriedade do heap máximo seja satisfeita. Desta forma, temos que a função `Heapify` possui complexidade $O(\log n)$, resultando em uma complexidade final de $O(n \log n)$ para o algoritmo do Heapsort.

Exemplo de funcionamento:

In [9]:
```
A = numpy.random.randint(-20, 20, 6)
print('Vetor de entrada: '+ str(A))
Heapsort(A)
print('Vetor ordenado: '+ str(A))
```

```
Vetor de entrada: [ 0 15  0  4  1 -6]
Vetor ordenado: [-6  0  0  1  4 15]
```