Part 1: Sorting algorithms

Introduction

The objective of a sorting algorithm it to put a list of values in either ascending or descending order. The efficiency of a sorting algorithm is evaluated by how little the process uses time and memory. In modern computers memory is not as much of an issue as time. This section of the report is meant to implement and then assess different sorting algorithms by comparing the duration they take to sort lists of randomly generated values from length 10 up to 10 000

Implementation

We will be implementing the following algorithms using python(3.9):

- Bubble sort
- Merge sort
- Quick sort
- Insertion sort
- Heap sort
- Selection sort

Bubble Sort

Introduction

The way bubble sort function is by repeatably going through the list and comparing each element with the next. Swapping them when necessary until it passes through th full list without the need to swap any elements. Meaning the list is fully sorted.

Implementation

In [7]:
```
def bubble_sort(array):
    sorted=False
    array_size=len(array)
    while(not sorted):
        sorted=True
        for i in range(array_size-1):
            if(array[i]>array[i+1]):
                array[i+1],array[i]=array[i],array[i+1]
                sorted=False
    return array
```

Complexity

The algorithm uses two loops one inside the other. The inner loop runs n times. The outer loop runs until the inner loop iterates without effecting any changes to the array; 1 time in the best case and n times in the worst case. Since we are only interested in the worst case complexity, that gives and overall complexity of:

- $O(n^2)$

Merge sort

Introduction

Merge sort is a divide and conquer algorithm that was invented by the famous physicians John Von Neumann in 1945. It functions by continually splitting the list into sub lists until they are small enough. Then merging each of the sub lists into a new sorted list.

Implementation

In [8]:
```
def merge_sort(array):
    length=len(array)
    if(length==1):
        return array
    ## Dividing the array
    array1=merge_sort(array[:length//2])
    array2=merge_sort(array[length//2:])

    ## Merging
    result_array=[]
    while(len(array1) and len(array2)):
        if(array1[0]<array2[0]):
            result_array.append(array1.pop(0))
        else:
            result_array.append(array2.pop(0))
    results_array=result_array+array1+array2
    return results_array
```

Complexity

The complexity of this implementations is $T(n) = 2T(n/2) + n$ using the master theorem we can conclude this function is of the complexity:

- $O(n \log n)$

Quick sort

Introduction

Quick sort is a sorting algorithm developed by Tony Hoare in 1959 and still in use today. It is another divide and conquer algorithm that functions by choosing a pivot then splitting the list into two depending on if the elements are greater or less then the pivot.

Implementation

In [9]:
```
def quick_sort(array):
    if(len(array)==1):
        return array
    ## sorting the array with one less element
    popped=array.pop()
    sorted_array=quick_sort(array)

    ## if the popped element the biggest in the sorted array
    length=len(sorted_array)
    if(popped>=sorted_array[length-1]):
        sorted_array.append(popped)
        return sorted_array

    ## Using binary search to find the right placement
    high,low=length-1,0
    while(low<high):
        mid=(high+low)//2
        if(popped<=sorted_array[mid]):
            high=mid
        else:
            low=mid+1

    sorted_array.insert(low,popped)
    return sorted_array
```

Complexity

This algorithm uses recursivety passing an array of n-1 size. And Binary search to find the right placement for the popped element in the sorted array. Therefore the complexity function $T(n) = T(n-1) + \log(n)$ which gives the overall complexity of:

- $O(n \log n)$

Many implementations of this algorithm are $O(n^2)$ instead. The way we reduced the complexity was the use of binary search instead of linear search in the insertion.

Insertion sort

Introduction

Insertion sort is a sorting algorithm that is so simple it could be implemented in three line in C++.

It functions by inserting each element in the already ordered section of the list one by one.

Implementation

In [10]:
```
def insertion_sort(array):
    length=len(array)
    for i in range(1,length):
        j=i
        while(j>0 and array[j-1]>array[j]):
            array[j-1],array[j]=array[j],array[j-1]
            j-=1
    return array
```

Complexity

The implementation uses outer loop that run n times. And an inner loop that run either 0 or n times. Which means this implementation has the complexity of:

- $O(n^2)$

Heap sort

Introduction

Heap sort is an improved version of selection sort. it is based on the binary heap data structure which is a binary tree where The descendent of each element are smaller then them.

Implementation

Our implementation uses a binary tree in the following way:

1. The root element is the first element of the list
2. For every index of the list i:
    - left child has index of 2i+1
    - right child has index of 2i+2

In [11]:
```
def heapify(array,length , root):
    ## creates a heap assuming the two subtrees are already a heap
    largest = root
    left  = 2*root+1
    right = 2*root+2
    if left < length and array[largest] < array[left]:
        largest = left
    if right < length and array[largest] < array[right]:
        largest = right
    if largest != root:
        array[root], array[largest] = array[largest], array[root]
        heapify(array, length, largest)


def heap_sort(array):
    length = len(array)
    heapify(array,length,1)
    ## creating the heap from the bottom up (igonring the leafs)
    for i in range(length//2 - 1, -1, -1):
        heapify(array, length, i)

    # Turning the heap into a sorted array
    for i in range(length-1, 0, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)
    return array
```

Complexity

Studying the complexity of this algorithm is a bit complicated. First heapify will in ht best case keep the root in it's place. Or in the worst case push it down to the leafs which means a it will run a maximum of $\log_2(n)$. which mean it's $O(\log n)$. For the sorting algorithm it runs the heapify function n/2 times when initializing the heap then n times in turning the heap back into a list. Therefore we can conclude The sorting algorithm is of the complexity:

- $O(n \log(n))$

Selection sort

Introduction

Selection sort is another simple sorting algorithm. It function by dividing the list into two parts. An ordered section that is built from the left to right. And an unordered section where the algorithm searches for the minimum value each time.

Implementation

In [12]:
```
def selection_sort(array):
    length=len(array)
    if (length == 1):
        return array
    min_index = 0

    for i in range(length):
        if (array[i] < array[min_index]):
            min_index = i

    min = array.pop(min_index)
    array = selection_sort(array)
    array.insert(0, min)
    return array
```

Complexity

This algorithm usees one loop that runs n time. And a callback with and array of one less element as well as an insertion function Which python implements in $O(1)$.So the complexity is $T(n) = T(n-1) + n$ which gives us an overall complexity of:

- $O(n^2)$

Collecting data

To study the time complexity of these functions. Qe created a function that logs the durations these algorithms take to sort a randomly generated set of lists. This function provides a few customizable parameters. Basically it will test starting from arrays of size `min_array_size` then increase by `step` until `max_array_size` . For each array size we will use a number of samples defined by `number_of_samples` to get as accurate of a reading as possible. We also provide an estimation of the time left as well as a percentage of the arrays tested while the function is running. Finally the results are saved into the two arrays `array_sizes` and `durations` which then can be saved to a CSV format file. In this project we provide the data collected in the data folder.

In [13]:
```
from random import random
from math import floor

import pandas as pd
import matplotlib.pyplot as plt
import time

def run_algo(sort_name, arr):
    '''Runs the named algorithm with the provided array'''
    if sort_name == "bubble":
        bubble_sort(arr)
    if sort_name == "quick":
        quick_sort(arr)
    if sort_name == "merge":
        merge_sort(arr)
    if sort_name == "insertion":
        insertion_sort(arr)
    if sort_name == "heap":
        heap_sort(arr)
    if sort_name == "selection":
        selection_sort(arr)


MAX_ARRAY_SIZE = 7_000
MIN_ARRAY_SIZE = 0
MAX_NUMBER = 1_000
NUMBER_OF_SAMPLES = 5
STEP = 10


def study(max_array_size=MAX_ARRAY_SIZE,
          min_array_size=MIN_ARRAY_SIZE,
          max_number=MAX_NUMBER,
          number_or_samples=NUMBER_OF_SAMPLES,
          step=STEP,
          algo_name="bubble",
          array_sizes=[],
          durations=[],
          ):
    execution_time = 0
    for array_size in range(max_array_size, min_array_size, -step):
        for i in range(number_or_samples, 0, -1):
            start = time.time()
            ## Estimating the duration left
            print(f' {floor((max_array_size-array_size)*100/max_array_si
                  f' ETA:{floor(number_or_samples*array_size*execution_t

            arr = [floor(max_number*random()) for _ in range(array_size)
            run_algo(algo_name, arr)
            end = time.time()
            execution_time = end-start
            array_sizes.append(array_size)
            durations.append(execution_time)
```

Example

We'll use Insertion Sort to demonstrate The study function. with array sizes up to 2000.

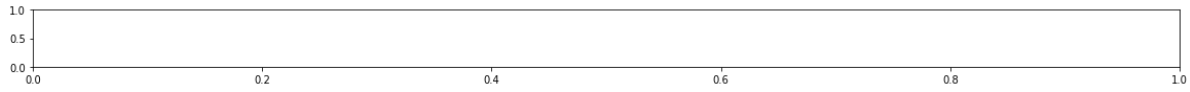
In [14]:
```
times = {}
arr_sizes = []
durations = []
study(array_sizes=arr_sizes,
      durations=durations,
      min_array_size=0,
      step=10,
      max_array_size=2000,
      algo_name="insertion",
        )

times["array sizes"] = arr_sizes
times["durations"] = durations
df = pd.DataFrame(times)
print(df)
```

```
     array sizes  durations
0           2000   0.763520
1           2000   0.711967
2           2000   0.653372
3           2000   0.823308
4           2000   0.668826
..           ...        ...
995           10   0.000069
996           10   0.000064
997           10   0.000062
998           10   0.000065
999           10   0.000059

[1000 rows x 2 columns]
```

We will use the matplotlib package to visualise the data in a scatter diagram.

In [15]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots()
plt.rcParams["figure.figsize"] = (20,1)
axes.scatter(df["array sizes"], df["durations"],label=f'{algo_name}',siz
plt.xlabel('Array size')
plt.ylabel('Executation time (ms)')
plt.legend(loc=2, prop={'size': 20},markerscale=6)
plt.show()
```

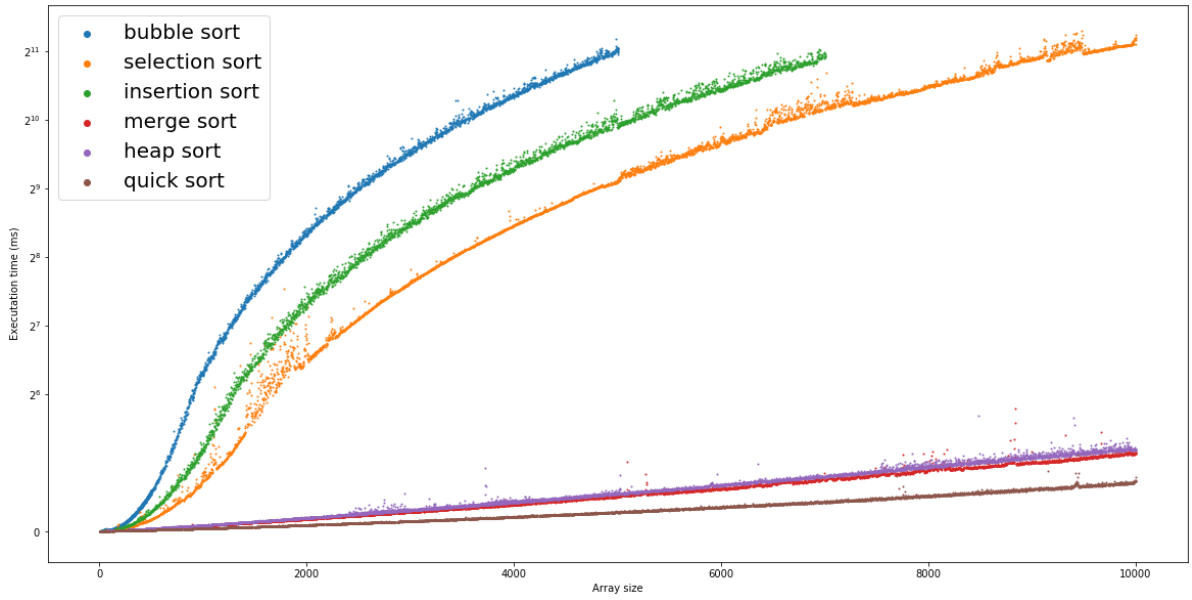
```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
 last)
<ipython-input-15-b59e4f49afad> in <module>
      3 fig, axes = plt.subplots()
      4 plt.rcParams["figure.figsize"] = (20,1)
----> 5 axes.scatter(df["array sizes"], df["durations"],label=f'{algo_
name}',sizes=[2])
      6 plt.xlabel('Array size')
      7 plt.ylabel('Executation time (ms)')

NameError: name 'algo_name' is not defined
```



The efficiency of each algorithmes

Using the study function we have pre compiled data on each algorithm on the same machine. Each algorithm is tested to atleast arrays of size 5000 up to 10 000. We will display our finding in a symlog graph. Meaning the start of the y axis is linear to avoid the small values disappearing into 0. And at specific threshold the y axis will switch to a logarithmic axis.

In [ ]:
```
sorts_data=["bubble","selection","insertion","merge","heap","quick"]
fig, axes = plt.subplots()
for sort_data in sorts_data:
    # read the data
    df = pd.read_csv(f'data/{sort_data}.csv')
    axes.scatter(df["array sizes"], [dur * 1000 for dur in df["durations
plt.legend(loc=2, prop={'size': 20},markerscale=6)
plt.rcParams["figure.figsize"] = (20,10)
plt.yscale('symlog',basey=2,basex=2
           ,linthreshy=2**6)
plt.xlabel('Array size')
plt.ylabel('Executation time (ms)')
plt.show()
```



We can see the effects git more evident the bigger the size of the array. for example quick sort is 100 times faster in arrays of size 10 000 then selection sort.

Part 2: Matrix Multiplication

Introduction

Matrix multiplication is a very central operation in many numerical algorithms. The implementation of the mathematical definition provides an algorithm with a complexity $O(n^2)$. However mathematicians and computer scientist have been able to develop algorithms that reduced that complexity. In 1969 Volger Strassen published his algorithm that reduced the complexity to $O(n^{2.373})$.

In this section we will implement both algorithms and compare them. As well as matrix multiplication function provided in the python library Numpy.

Naive

The naive matrix multiplication algorithm uses three loop each one runs n times. Giving the overall complexity of $O(n^2)$.

In [ ]:
```
def classic_matrix_mult(A, B):
    n=len(A)
    R=[[0]*n]*n
    for i in range(n):
        for j in range(n):
            for k in range(n):
                R[i][j] += A[i][k]*B[i][k]
    return R
```

Strassens algorithm

To implement Strassens algorithm, first we need a few helper functions `add`, `substract` and `divide`. The first two allow to do basic arithmetics on matrixes. The last one will divide the matrix into 4 sub matrices.

In [ ]:
```
def add(A, B):
    n=len(A)
    R=[[0]*n]*n
    return[[A[i][j]+B[i][j] for j in range(n)] for i in range(n)]

def subtract(A, B):
    n=len(A)
    return[[A[i][j]-B[i][j] for j in range(n)] for i in range(n)]


def divide(A):
    n = len(A)
    m = n // 2
    A11 = [[A[i][j] for j in range(m)] for i in range(m)]
    A12 = [[A[i][j] for j in range(m)] for i in range(m, n)]
    A21 = [[A[i][j] for j in range(m, n)] for i in range(m)]
    A22 = [[A[i][j] for j in range(m, n)] for i in range(m, n)]
    return A11,A12,A21,A22
```

Then we will implement the actual algorithm.

In [ ]:
```
def strassen_matrix_mult(A, B,threshehold=512):
    if len(A) <=threshehold:
        C =classic_matrix_mult(A,B)

    else:
        A11, A12, A21, A22 = divide(A)
        B11, B12, B21, B22 = divide(B)

        s1 = strassen_matrix_mult(A11, subtract(B12, B22))
        s2 = strassen_matrix_mult(add(A11, A12), B22)
        s3 = strassen_matrix_mult(add(A21, A22), B11)
        s4 = strassen_matrix_mult(A22, subtract(B21, B11))
        s5 = strassen_matrix_mult(add(A11, A22), add(B11, B22))
        s6 = strassen_matrix_mult(subtract(A12, A22), add(B21, B22))
        s7 = strassen_matrix_mult(subtract(A11, A21), add(B11, B12))

        C11 = add(subtract(add(s5, s4), s2), s6)
        C12 = add(s1, s2)
        C21 = add(s3, s4)
        C22 = subtract(subtract(add(s1, s5), s3), s7)

        C = []
        for i in range(len(C12)):
            C.append(C11[i] + C12[i])
        for i in range(len(C22)):
            C.append(C21[i] + C22[i])

    return C
```

We chose a relatively high number to switch the naive algorithm. This is because while strassens algorithm reduces the number of multiplications The overhead of array allocation, additions and subtractions outweighs it's benefits in small enough matrices.

Numpy

Numpy is a python library that provides support for large multi-dimensional array and matrices. along with high-level mathematical functions to operate those arrays. We will be using it's `dot` function that allows for matrices multiplication.

In [ ]:
```
import numpy as np
def numpy_mult(A, B):
    return np.dot(A, B)
```

Comparison

First we created a function that generates matrices with random elements.

In [ ]:
```
import random

def gen_matrix(n):
    matrice = []
    for i in range(n):
        matrice.append([random.randrange(0, 9) for i in range(n)])
    return matrice
# Generating the random matrices.
sizes = [64, 128, 256, 512,1024,2048,4096]
A = []
for i in range(len(sizes)):
    A.append(gen_matrix(sizes[i]))
B = []
for i in range(len(sizes)):
    B.append(gen_matrix(sizes[i]))
```

We also created a function that logs the duration it takes for each function to do the multiplication.

In [ ]:
```
def duration(A,B,function):
     start = time.time()
    #  print(function(A,B))
     function(A,B)
     end = time.time()
     temps = end - start
     print({len(A)},"=>",temps)
     return temps
```

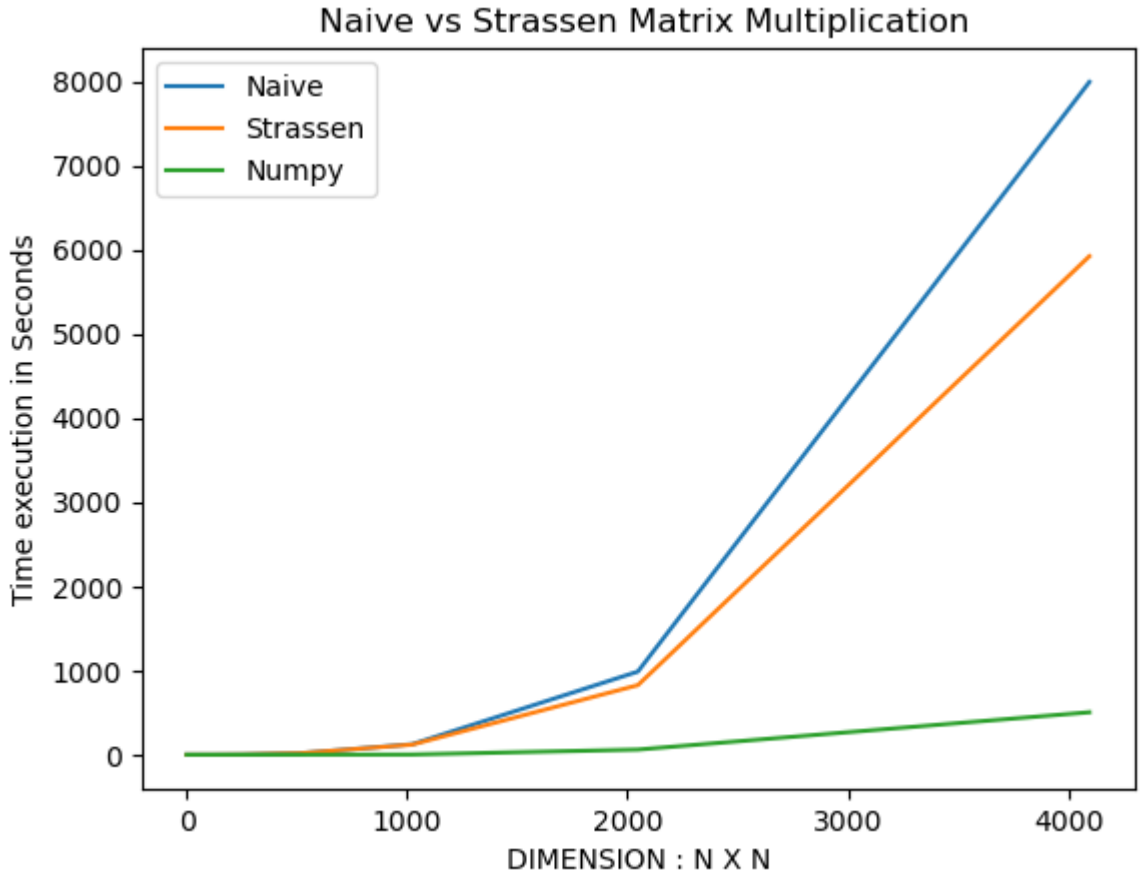
Finally we will run all the algorithm and collect their data and draw the graphs.

In [ ]:
```
# Strassen
print("strassen_matrix_mult")
durations2 = []
for i in range(len(sizes)):
    durations2.append(duration(A[i], B[i], strassen_matrix_mult))

# Classic/naive
print("classic_matrix_mult")
durations1 = []
for i in range(len(sizes)):
    durations1.append(duration(A[i], B[i], classic_matrix_mult))

# Numpy.dot
print("Numpy")
durations3 = []
for i in range(len(sizes)):
    durations3.append(duration(A[i], B[i], numpy mult))
```

In [ ]:
```
# Drawing the graph.
plt.title(" Naive vs Strassen Matrix Multiplication")
plt.xlabel('DIMENSION : N X N')
plt.ylabel('Time execution in Seconds')
plt.plot(sizes,durations1, label='Naive')
plt.plot(sizes,durations1 , label='Strassen')
plt.plot(sizes,durations1 , label='Numpy')
plt.legend()
plt.rcParams["figure.figsize"] = (20,3)
plt.show()
#8094 1019 131 16
```

```
classic_matrix_mult
le temps d'execution  pour la matrice de taille  {2} => 4.76837158203
125e-06
le temps d'execution  pour la matrice de taille  {4} => 1.09672546386
71875e-05
le temps d'execution  pour la matrice de taille  {8} => 6.43730163574
2188e-05
le temps d'execution  pour la matrice de taille  {16} => 0.0004692077
63671875
le temps d'execution  pour la matrice de taille  {32} => 0.0036401748
657226562
le temps d'execution  pour la matrice de taille  {64} => 0.0288186073
30322266
le temps d'execution  pour la matrice de taille  {128} => 0.228962898
25439453
le temps d'execution  pour la matrice de taille  {256} => 1.848454236
984253
le temps d'execution  pour la matrice de taille  {512} => 15.39847993
850708
le temps d'execution  pour la matrice de taille  {1024} => 123.454937
45803833
le temps d'execution  pour la matrice de taille  {2048} => 987.152979
850769
le temps d'execution  pour la matrice de taille  {4096} => 7991.54085
7076645
Numpy
le temps d'execution  pour la matrice de taille  {2} => 2.74181365966
79688e-05
le temps d'execution  pour la matrice de taille  {4} => 1.14440917968
75e-05
le temps d'execution  pour la matrice de taille  {8} => 2.36034393310
54688e-05
le temps d'execution  pour la matrice de taille  {16} => 0.0001997947
6928710938
le temps d'execution  pour la matrice de taille  {32} => 0.0002036094
6655273438
le temps d'execution  pour la matrice de taille  {64} => 0.0007607936
859130859
le temps d'execution  pour la matrice de taille  {128} => 0.004089355
46875
le temps d'execution  pour la matrice de taille  {256} => 0.030673503
875732422
le temps d'execution  pour la matrice de taille  {512} => 0.216464519
50073242
le temps d'execution  pour la matrice de taille  {1024} => 1.83395266
```

```
53289795
le temps d'execution  pour la matrice de taille  {2048} => 60.6507918
8346863
le temps d'execution  pour la matrice de taille  {4096} => 503.790102
24342346
```



le temps d'execution pour la matrice de taille {2047} => 826.056037902832 le temps d'execution pour la matrice de taille {4095} => 5921.243574619293 classic_matrix_mult le temps d'execution pour la matrice de taille {1} => 6.9141387939453125e-06 le temps d'execution pour la matrice de taille {3} => 1.1205673217773438e-05 le temps d'execution pour la matrice de taille {7} => 6.508827209472656e-05 le temps d'execution pour la matrice de taille {15} => 0.0004801750183105469 le temps d'execution pour la matrice de taille {31} => 0.003806591033935547 le temps d'execution pour la matrice de taille {63} => 0.03067183494567871 le temps d'execution pour la matrice de taille {127} => 0.24541163444519043 le temps d'execution pour la matrice de taille {255} => 1.9685826301574707 le temps d'execution pour la matrice de taille {511} => 15.696885347366333 le temps d'execution pour la matrice de taille {1023} => 125.75883078575134 le temps d'execution pour la matrice de taille {2047} => 1001.6408014297485 le temps d'execution pour la matrice de taille {4095} => 8213.931849479675 Numpy le temps d'execution pour la matrice de taille {1} => 8.96453857421875e-05 le temps d'execution pour la matrice de taille {3} => 1.239776611328125e-05 le temps d'execution pour la matrice de taille {7} => 2.9087066650390625e-05 le temps d'execution pour la matrice de taille {15} => 6.103515625e-05 le temps d'execution pour la matrice de taille {31} => 0.00020122528076171875 le temps d'execution pour la matrice de taille {63} => 0.0008134841918945312 le temps d'execution pour

la matrice de taille {127} => 0.003866910934448242 le temps d'execution pour la matrice de taille {255} => 0.029384136199951172 le temps d'execution pour la matrice de taille {511} => 0.21466970443725586 le temps d'execution pour la matrice de taille {1023} => 1.8973238468170166 le temps d'execution pour la matrice de taille {2047} => 62.02652454376221 le temps d'execution pour la matrice de taille {4095} => 516.6746072769165