Farhat Abbas
Sciences Faculty

TP1 AAC Report

Prepared by

Rami Boukaroura G9
Maâbed Amina G8

2022/2023

Part 1: Sorting Algorithms

Introduction

The objective of a sorting algorithm is 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 functions is by repeatably going through the list and comparing each element with the next. Swapping them when necessary until it passes through the full list without the need to swap any elements, meaning the list is fully sorted.

Implementation

In [1]:
```
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 affecting 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 an overall complexity of

- $O(n^2)$

Merge sort

Introduction

Merge sort is a divide-and-conquer algorithm that was invented by the famous physicist 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 [2]:
```
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 implementation 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 is still in use today. It is another divide-and-conquer algorithm that functions by choosing a pivot and then splitting the list into two, depending on if the elements are greater or less than the pivot.

Implementation

In [3]:
```
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 recursivity 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. through the use of binary search instead of linear search we were able to reduce the complexity

Insertion sort

Introduction

Insertion sort is a sorting algorithm that is so simple it could be implemented in three lines in

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

Implementation

In [4]:
```
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 an outer loop that runs n times. And an inner loop that runs either 0 or n times. This means this implementation has the complexity of

- $O(n^2)$

Heap sort

Introduction

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 node of the list:
    - left child has index of 2i+1
    - right child has index of 2i+2

In [5]:
```
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, in the best case heapify will keep the root in its place. Or in the worst case, push it down to the leaves which means it will run a maximum of $\log_2(n)$, which means it's $O(\log n)$. For the sorting algorithm, it runs the heapify function $\frac{n}{2}$ 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 functions by dividing the list into two parts an ordered section and unordered section. The ordered section is built from left to right and an unordered section where the algorithm searches for the minimum value each time.

Implementation

In [6]:

```
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 uses one loop that runs n time and a callback with an 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, we 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` and 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 [7]:
```
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(
          algo_name,
          max_array_size=MAX_ARRAY_SIZE,
          min_array_size=MIN_ARRAY_SIZE,
          max_number=MAX_NUMBER,
          number_or_samples=NUMBER_OF_SAMPLES,
          step=STEP,

          ):
    times = {}
    execution_time = 0
    array_sizes=[]
    durations=[]
    for array_size in range(max_array_size, min_array_size, -step):
        for i in range(number_or_samples, 0, -1):

            start = time.time()
            ## Printing an estimation of the duration left
            print(f' {floor((max_array_size-array_size)*100/max_array_si
                  +f' ETA:{floor(number_or_samples*array_size*execution_
                  end='  \r')

            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)
    times["array sizes"] = array_sizes
    times["durations"] = durations
    return times
```

Example

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

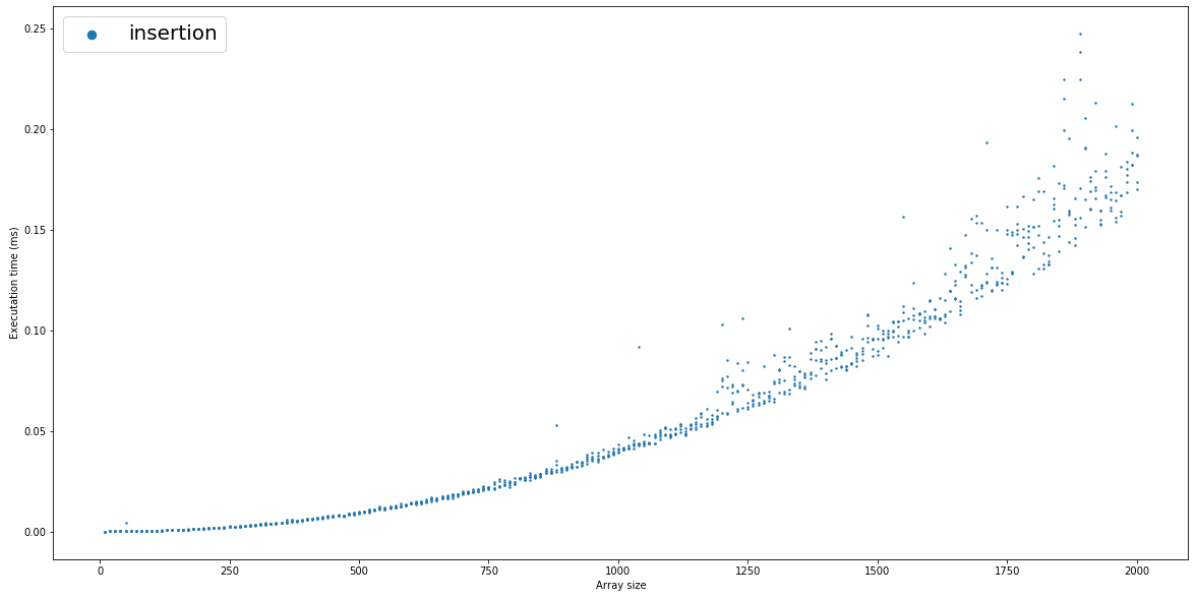
In [8]:

```
arr_sizes = []
durations = []
times=study(
      min_array_size=0,
      step=10,
      max_array_size=2_000,
      algo_name="insertion",
        )


df = pd.DataFrame(times)
print(df)
```

```
     array sizes  durations
0           2000   0.186748
1           2000   0.196146
2           2000   0.187378
3           2000   0.170238
4           2000   0.173597
..           ...        ...
995           10   0.000018
996           10   0.000017
997           10   0.000019
998           10   0.000016
999           10   0.000018

[1000 rows x 2 columns]
```

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

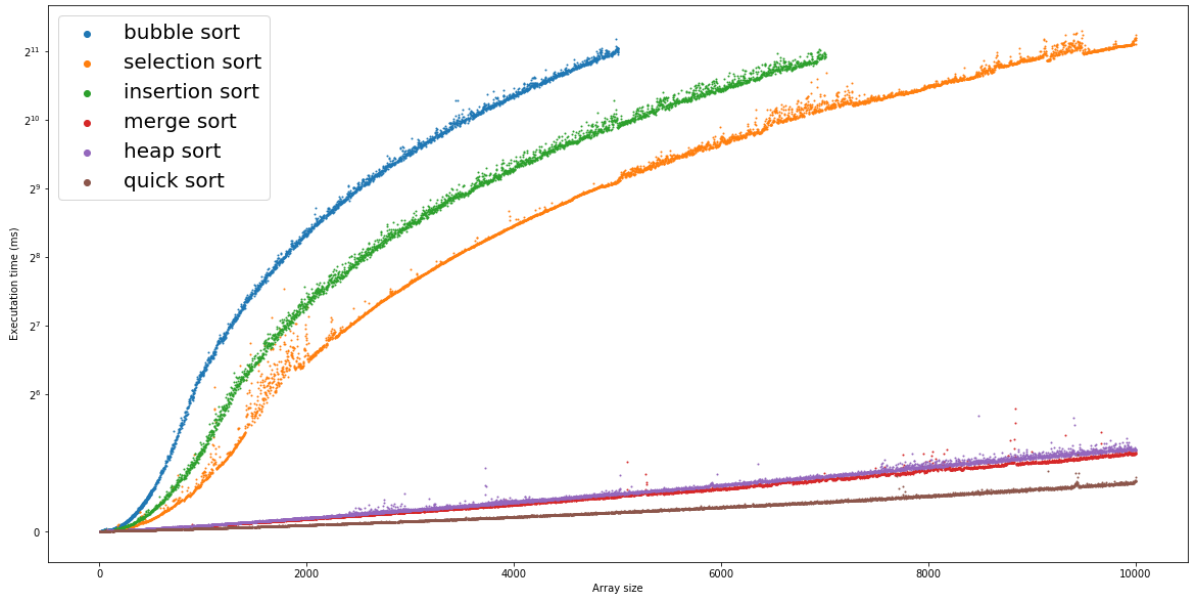
In [10]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots()
plt.rcParams["figure.figsize"] = (20,10)
axes.scatter(df["array sizes"], df["durations"],label='insertion',sizes=
plt.xlabel('Array size')
plt.ylabel('Executation time (ms)')
plt.legend(loc=2, prop={'size': 20},markerscale=6)
plt.show()
```



The efficiency of the algorithmes

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

In [11]:
```
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"]]
                 ,label=f'{sort_data} sort',sizes=[1])
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 get more evident the bigger the array size. For example, quick sort is 100 times faster in arrays of size 10 000 than 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 complexity $O(n^2)$. However, mathematicians and computer scientists have been able to develop algorithms that

reduced that complexity. In 1969 Volger Strassen published his algorithm that reduced the complexity to $O(n^{2.8074})$.

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

Naive

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

In [12]:
```
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
```

Strassen

To implement Strassen's algorithm, first, we need a few helper functions `add`, `substract`, and `divide`. The first two allow doing basic arithmetics on matrices. The last one will divide the matrix into 4 sub-matrices.

In [13]:
```
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 [14]:
```
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 to the naive algorithm. This is because while Strassen's algorithm reduces the number of multiplications The overhead of array allocation, additions and subtractions outweigh its 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 its `dot` function that allows for matrices multiplication.

In [15]:
```
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 [17]:
```
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 = [2,4,8,16,32,64, 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 few functions that help log the duration it takes for each function to do the multiplication.

In [18]:
```
def mult(A,B,function_name):
    if function_name=="strassen":
        strassen_matrix_mult(A,B)
    if function_name=="classic":
        classic_matrix_mult(A,B)
    if function_name=="numpy":
        np.dot(A,B)


def duration(A,B,function_name):
     start = time.time()
    #  print(function(A,B))
     mult(A,B,function_name)
     end = time.time()
     temps = end - start
     return temps

def study(A,B,function_name,sizes):
    durations = []
    for i in range(len(sizes)):
        durations.append(duration(A[i],B[i],function_name))
    return durations
```

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

In [19]:
```
times={}
times["matrix sizes"]=sizes
times["durations"]=[]
algo_names=["classic","strassen","numpy"]

for algo_name in algo_names:
    print(f"Studying: {algo_name}")
    times["durations"]=study(A,B,algo_name,sizes)
    df=pd.DataFrame(times)
    df.to_csv(f"data/.{algo_name}.csv")
```

```
Studying: classic
Studying: strassen
Studying: numpy
```

In [21]:
```
# Drawing the graph.
for algo_name in algo_names:
    # read the data
    df = pd.read_csv(f'data/{algo_name}.csv')
    plt.plot(df["matrix sizes"], df["durations"],label=f'{algo_name} sor
plt.legend(loc=2, prop={'size': 20},markerscale=6)
plt.rcParams["figure.figsize"] = (20,10)
plt.title(" Naive vs Strassen Matrix Multiplication")
plt.xlabel('Matrix size')
plt.ylabel('Time execution in Seconds')
plt.rcParams["figure.figsize"] = (20,10)
plt.show()
```