BTH004 - Laboratory assignment 2—— Theoretical analysis and Practical analysis between merge sort and bubble sort

name: Zhao Shui–
student ID: 201806150329

November 2020

1 description of two algorithm

1.1 merge sort

Use the divide-and-conquer algorithm. divide the original array into two parts, and for each parts redo this action until it can't be divided. and it become n parts(n is the size of the array), and then continuously combine the n parts number in order to the original size.

1.2 bubble sort

Use two for loop, it iteratively visit the element of the array to be sorted, compare the two adjacent element in turn. If these two elements are in the wrong order then swap them.

2 pseudo code for two algorithm

2.1 merge sort

```
def merge_sort(A):
    if len(A) <= 1:
        return A
    else:
        middle = len(A) // 2
        LA = A[:middle]
        RA = A[middle:]
        LA = merge_sort(LA)
        RA = merge_sort(RA)
        SA = merge(LA,RA)
        return SA

def merge(LA,RA):
```

```
    SA = []
    l = 0
    r = 0
    while l < len(LA) and r < len(RA):
        if LA[l] <= RA[r]:
            SA.append(LA[l])
            l += 1
        elif LA[l] >= RA[r]:
            SA.append(RA[r])
            r += 1
    SA += LA[l:]
    SA += RA[r:]
    return SA
```

2.2 bubble sort

```
def bubble_sort(A):
    for i in range(1,len(A)-1):
        for j in range(len(A)-i):
            temp = A[j]
            if A[j] > A[j + 1]:
                A[j] = A[j+1]
                A[j+1] = temp
    return A
```

3 code for the experiment

```
import random
import matplotlib.pyplot as plt
import time
import numpy as np

def bubbleSort(A):
    for i in range(1,len(A)-1):
        for j in range(len(A)-i):
            temp = A[j]
            if A[j] > A[j + 1]:
                A[j] = A[j+1]
                A[j+1] = temp
    return A

def merge_sort(A):
    if len(A) <= 1:
        return A
    else:
        middle = len(A) // 2
        LA = A[:middle]
        RA = A[middle:]
        LA = merge_sort(LA)
        RA = merge_sort(RA)
        SA = merge(LA,RA)
        return SA

def merge(LA,RA):
    SA = []
    l = 0
```

```
    r = 0

    while l < len(LA) and r < len(RA):
        if LA[l] <= RA[r]:
            SA.append(LA[l])
            l += 1
        elif LA[l] >= RA[r]:
            SA.append(RA[r])
            r += 1
    SA += LA[l:]
    SA += RA[r:]
    return SA

def createRandomArray(numOfArray,numRange):
    arr = []
    for i in range(0,numOfArray):
        arr.append(random.randint(0, numRange))
    return arr

PowerArray = []
Time = []

power = 500
step = power
amount = 40

while power <= step * amount:
    PowerArray.append(power)
    TimeTemp1 = []
    TimeTemp2 = []
    fmerge = np.log(power) * power - power + 1
    fbubble =  power * (power - 1) /2
    for j in range(0,50):

        oldT = time.time()
        array = createRandomArray(power, power)
        newT = time.time()

        oldtime = time.time()
        merge_sort(array)
        newtime = time.time()

        TimeTemp1.append(newtime-oldtime)
        TimeTemp2.append(newT-oldT)

    Time.append(np.average(TimeTemp1) / fmerge)
    rate = np.average(TimeTemp2) / np.average(TimeTemp1) + np.
    average(TimeTemp2)

    print('array size: {} initialization time: {}%'.format(power,
    rate * 100 ))
    power += step

plt.xlabel("array size")
plt.ylabel("c")
plt.plot(PowerArray, Time)
plt.show()
```

4 theoretical analysis

4.1 merge sort

the number of merge sort is:

$$f(n) = n \log n - n + 1$$

4.2 bubble sort

the number of bubble sort is:

$$f(n) = \frac{(n-1)n}{2}$$

5 theoretical analysis

5.1 merge sort

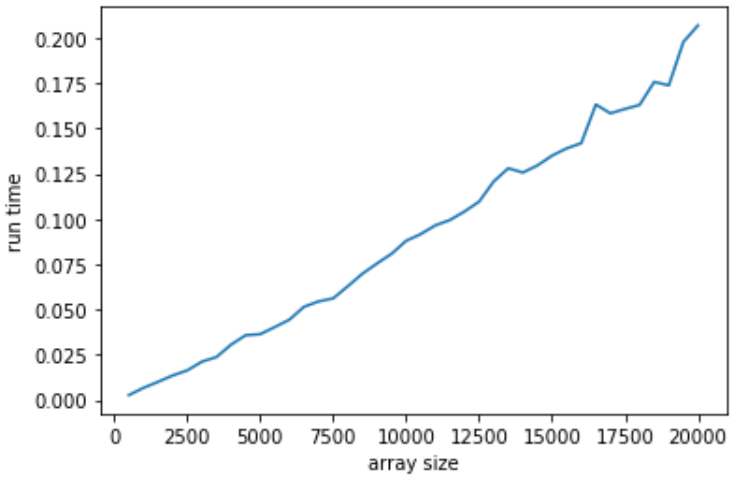

Figure 1: time of merge sort at different size of array

5.2 bubble sort

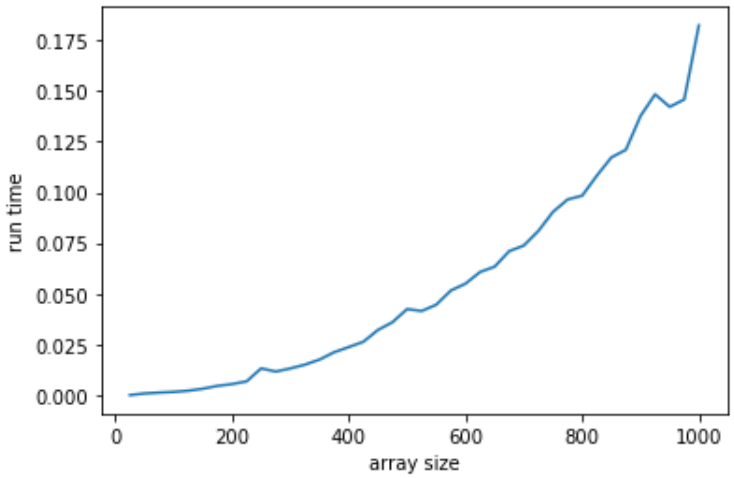

Figure 2: time of bubble sort at different size of array

6 initialization time at difference array size

6.1 merge sort

```
array size: 11000 ,initialization time 20.358436382772712%
array size: 11500 ,initialization time 19.84661101249%
array size: 12000 ,initialization time 19.846427402490242%
array size: 12500 ,initialization time 20.26128713225889%
array size: 13000 ,initialization time 20.333918721743384%
array size: 13500 ,initialization time 20.402474627025217%
array size: 14000 ,initialization time 20.413127464789486%
array size: 14500 ,initialization time 19.74109437763326%
array size: 15000 ,initialization time 19.940376774010907%
array size: 15500 ,initialization time 19.72761388636358%
array size: 16000 ,initialization time 19.360171081451185%
array size: 16500 ,initialization time 21.696202392890065%
array size: 17000 ,initialization time 21.397697755828887%
array size: 17500 ,initialization time 20.661082012828132%
array size: 18000 ,initialization time 20.874725494330775%
array size: 18500 ,initialization time 21.493588479481023%
array size: 19000 ,initialization time 21.501341634627423%
array size: 19500 ,initialization time 20.993321124995983%
array size: 20000 ,initialization time 22.195457169258535%
```

Figure 3: initialization time at different size of array

And we can find that it almost doesn't have any change when array size changed.

6.2 bubble sort

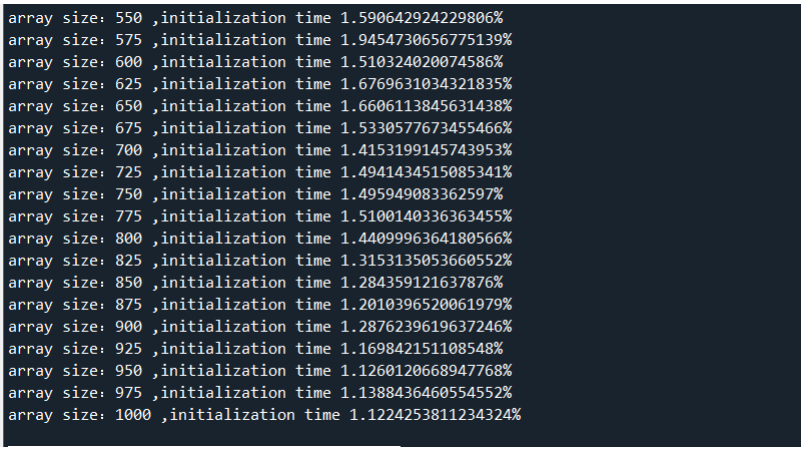

Figure 4: initialization time at different size of array

And we can obviously find that when array size increasing, initialization time is decreasing. And when array size come to 1000 or larger, the time is only 1%.

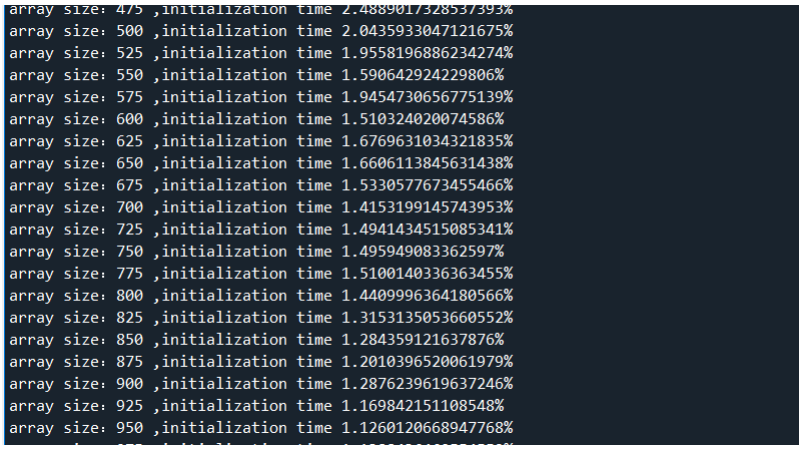

Figure 5: initialization time at different size of array

7 the comparison C of practical and theoretical analyses

7.1 merge sort

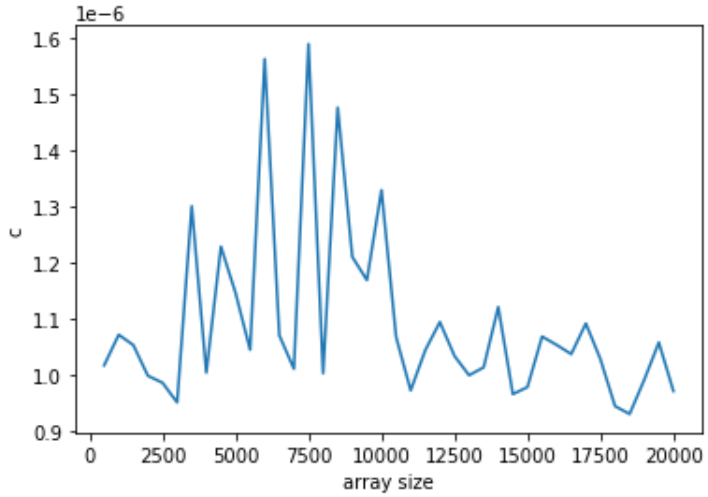

Figure 6: C of merge sort at different size of array

7.2 bubble sort

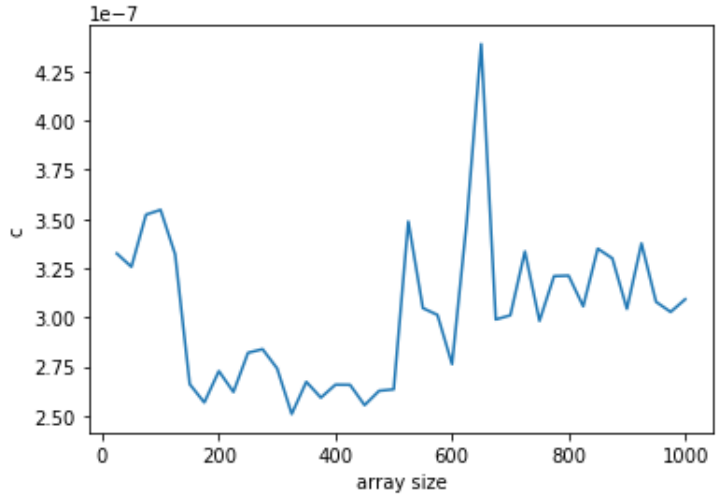

Figure 7: C of bubble sort at different size of array