

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

```
1 def merge_sort(A):
2     if len(A) <= 1:
3         return A
4     else:
5         middle = len(A) // 2
6         LA = A[:middle]
7         RA = A[middle:]
8         LA = merge_sort(LA)
9         RA = merge_sort(RA)
10        SA = merge(LA, RA)
11        return SA
12
13 def merge(LA, RA):
```

```

14 SA = []
15 l = 0
16 r = 0
17 while l < len(LA) and r < len(RA):
18     if LA[l] <= RA[r]:
19         SA.append(LA[l])
20         l += 1
21     elif LA[l] >= RA[r]:
22         SA.append(RA[r])
23         r += 1
24 SA += LA[l:]
25 SA += RA[r:]
26 return SA

```

2.2 bubble sort

```

1 def bubble_sort(A):
2     for i in range(1, len(A)-1):
3         for j in range(len(A)-i):
4             temp = A[j]
5             if A[j] > A[j + 1]:
6                 A[j] = A[j+1]
7                 A[j+1] = temp
8     return A

```

3 code for the experiment

```

1 import random
2 import matplotlib.pyplot as plt
3 import time
4 import numpy as np
5
6 def bubbleSort(A):
7     for i in range(1, len(A)-1):
8         for j in range(len(A)-i):
9             temp = A[j]
10            if A[j] > A[j + 1]:
11                A[j] = A[j+1]
12                A[j+1] = temp
13    return A
14
15 def merge_sort(A):
16     if len(A) <= 1:
17         return A
18     else:
19         middle = len(A) // 2
20         LA = A[:middle]
21         RA = A[middle:]
22         LA = merge_sort(LA)
23         RA = merge_sort(RA)
24         SA = merge(LA, RA)
25         return SA
26
27 def merge(LA, RA):
28     SA = []
29     l = 0

```

```

30     r = 0
31
32     while l < len(LA) and r < len(RA):
33         if LA[l] <= RA[r]:
34             SA.append(LA[l])
35             l += 1
36         elif LA[l] >= RA[r]:
37             SA.append(RA[r])
38             r += 1
39     SA += LA[l:]
40     SA += RA[r:]
41     return SA
42
43 def createRandomArray(numOfArray, numRange):
44     arr = []
45     for i in range(0, numOfArray):
46         arr.append(random.randint(0, numRange))
47     return arr
48
49 PowerArray = []
50 Time = []
51
52 power = 500
53 step = power
54 amount = 40
55
56 while power <= step * amount:
57     PowerArray.append(power)
58     TimeTemp1 = []
59     TimeTemp2 = []
60     fmerge = np.log(power) * power - power + 1
61     fbubble = power * (power - 1) / 2
62     for j in range(0, 50):
63
64         oldT = time.time()
65         array = createRandomArray(power, power)
66         newT = time.time()
67
68         oldtime = time.time()
69         merge_sort(array)
70         newtime = time.time()
71
72         TimeTemp1.append(newtime - oldtime)
73         TimeTemp2.append(newT - oldT)
74
75     Time.append(np.average(TimeTemp1) / fmerge)
76     rate = np.average(TimeTemp2) / np.average(TimeTemp1) + np.
77         average(TimeTemp2)
78     print('array size: {} initialization time: {}'.format(power,
79         rate * 100 ))
80     power += step
81
82 plt.xlabel("array size")
83 plt.ylabel("c")
84 plt.plot(PowerArray, Time)
85 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

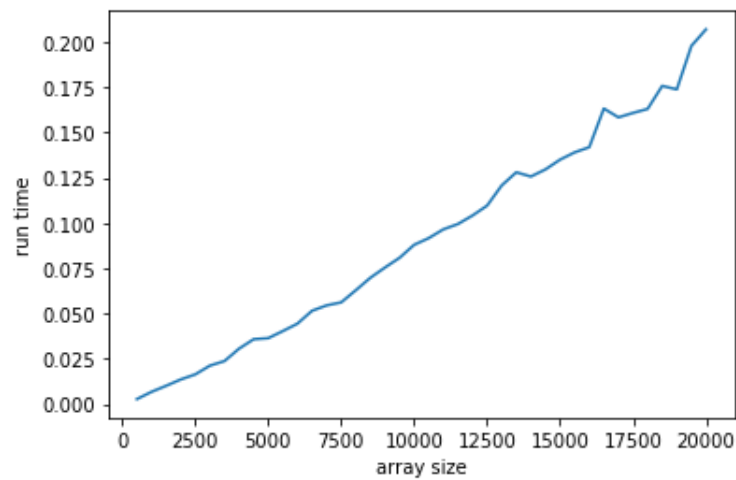


Figure 1: time of merge sort at different size of array

5.2 bubble sort

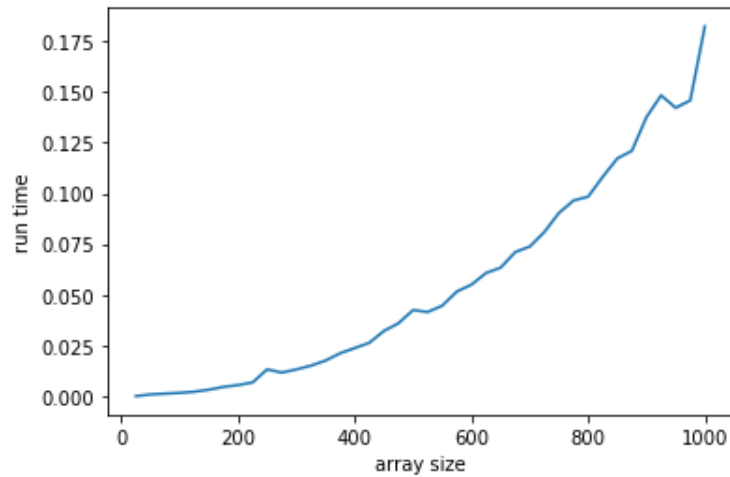


Figure 2: time of bubble sort at different size of array

6 initialization time at difference array size

6.1 merge sort

```
array size: 11000 ,initialization time 20.358436382772712%
array size: 11500 ,initialization time 19.846611012249%
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

```
array size: 550 ,initialization time 1.590642924229806%
array size: 575 ,initialization time 1.9454730656775139%
array size: 600 ,initialization time 1.510324020074586%
array size: 625 ,initialization time 1.6769631034321835%
array size: 650 ,initialization time 1.6606113845631438%
array size: 675 ,initialization time 1.5330577673455466%
array size: 700 ,initialization time 1.4153199145743953%
array size: 725 ,initialization time 1.4941434515085341%
array size: 750 ,initialization time 1.495949083362597%
array size: 775 ,initialization time 1.5100140336363455%
array size: 800 ,initialization time 1.4409996364180566%
array size: 825 ,initialization time 1.3153135053660552%
array size: 850 ,initialization time 1.284359121637876%
array size: 875 ,initialization time 1.2010396520061979%
array size: 900 ,initialization time 1.2876239619637246%
array size: 925 ,initialization time 1.169842151108548%
array size: 950 ,initialization time 1.1260120668947768%
array size: 975 ,initialization time 1.1388436460554552%
array size: 1000 ,initialization time 1.1224253811234324%
```

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%.

```
array size: 475 ,initialization time 2.4889017328537393%
array size: 500 ,initialization time 2.0435933047121675%
array size: 525 ,initialization time 1.9558196886234274%
array size: 550 ,initialization time 1.590642924229806%
array size: 575 ,initialization time 1.9454730656775139%
array size: 600 ,initialization time 1.510324020074586%
array size: 625 ,initialization time 1.6769631034321835%
array size: 650 ,initialization time 1.6606113845631438%
array size: 675 ,initialization time 1.5330577673455466%
array size: 700 ,initialization time 1.4153199145743953%
array size: 725 ,initialization time 1.4941434515085341%
array size: 750 ,initialization time 1.495949083362597%
array size: 775 ,initialization time 1.5100140336363455%
array size: 800 ,initialization time 1.4409996364180566%
array size: 825 ,initialization time 1.3153135053660552%
array size: 850 ,initialization time 1.284359121637876%
array size: 875 ,initialization time 1.2010396520061979%
array size: 900 ,initialization time 1.2876239619637246%
array size: 925 ,initialization time 1.169842151108548%
array size: 950 ,initialization time 1.1260120668947768%
```

Figure 5: initialization time at different size of array

7 the comparison C of practical and theoretical analyses

7.1 merge sort

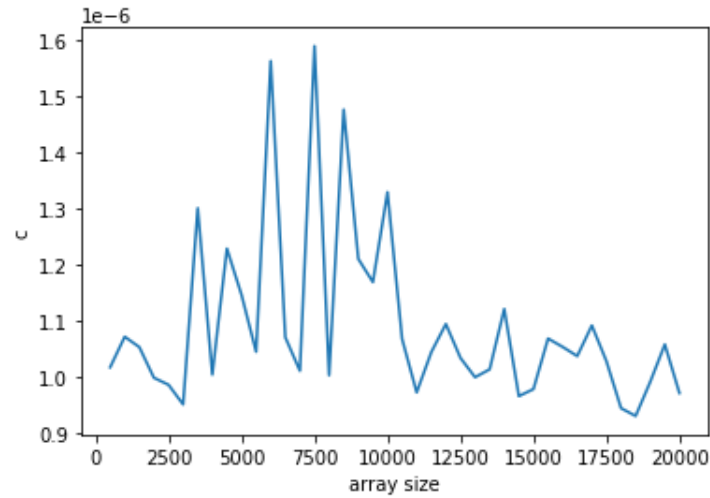


Figure 6: C of merge sort at different size of array

7.2 bubble sort

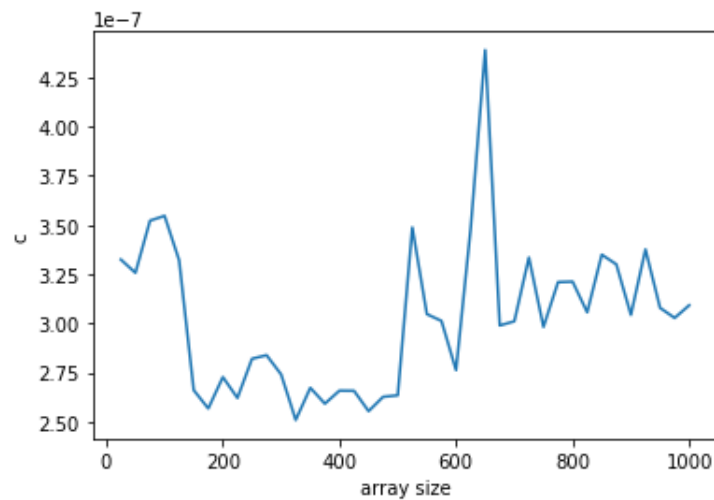


Figure 7: C of bubble sort at different size of array