report2

1 Merge sort and Quick sort

1.1 Plot of the two algorithms vs. size of the input array

Note that in the quick sort, the pivot element is set to be the rightmost element of the input array. Under this setting, if the input array is already sorted, we would expect the quick sort falls into the worst case complexity, which is $O(n^2)$. More details can be found below.

1.1.1 When the input arrays are randomly sorted

[2]:
```
from hw2 import *
import numpy as np
import pandas as pd
import time
import math
import matplotlib.pyplot as plt
```

[3]:
```
runtime_merge =[]
runtime_quick =[]
for n in range(7):
    array = np.random.randint(low=-1e4, high=1e4, size=(10**n,))
    array_c = array.copy()
    # merge sort
    st1 = time.time()
    merge_sort(array, 0, len(array) - 1)
    et1 = time.time()
    runtime_merge.append(et1-st1)
    # quick sort
    st2 = time.time()
    quickSort(array_c, 0, len(array_c) - 1)
    et2 = time.time()
    runtime_quick.append(et2-st2)
```

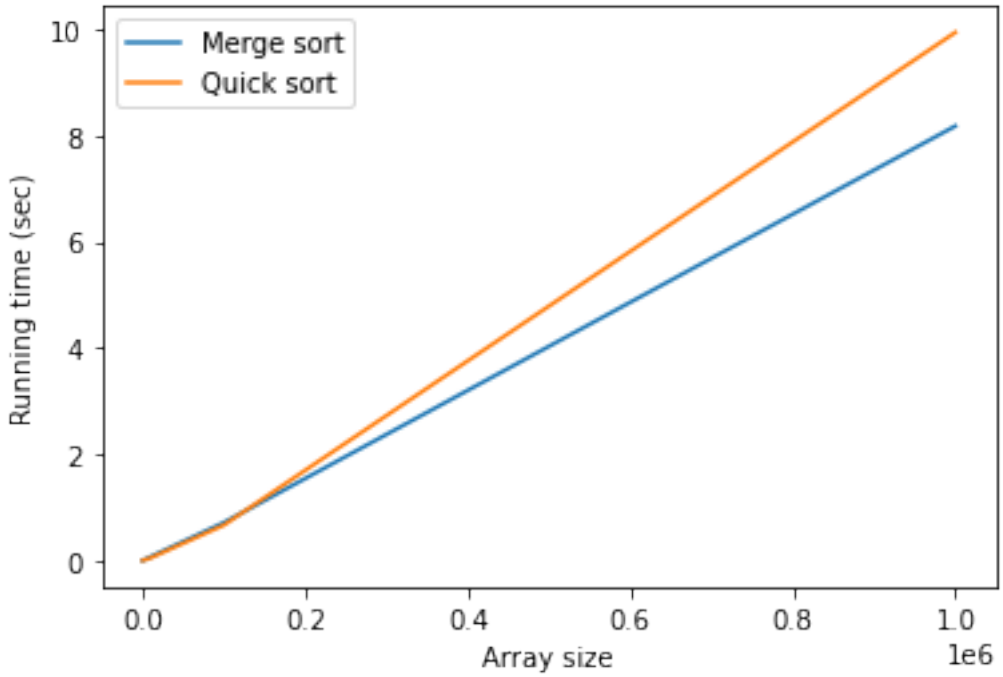
[7]:
```
size_n = [10**i for i in range(7)]
theory_merge = [math.floor(i*math.log(i)) for i in size_n]
theory_quick_worst = [i*i for i in size_n]
theory_quick_average = [math.floor(i*math.log(i)) for i in size_n]
```

```
data = {'size': size_n,
        'merge sort' : runtime_merge,
        'quick sort' : runtime_quick,
        'T(n) merge' : theory_merge,
        'T(n) quick(worst)' : theory_quick_worst,
        'T(n) quick(average)' : theory_quick_average}
df = pd.DataFrame(data)
print(df)
```

```
      size  merge sort    quick sort  T(n) merge  T(n) quick(worst)  \
0        1    0.000003  9.536743e-07           0                  1
1       10    0.000046  2.789497e-05          23                100
2      100    0.000534  3.960133e-04         460              10000
3     1000    0.006944  6.429195e-03        6907            1000000
4    10000    0.071774  5.060983e-02       92103          100000000
5   100000    0.714029  6.666191e-01     1151292        10000000000
6  1000000    8.171192  9.932312e+00    13815510      1000000000000

   T(n) quick(average)
0                    0
1                   23
2                  460
3                 6907
4                92103
5              1151292
6             13815510
```

[8]:
```
plt.plot( data['size'], data['merge sort'], label = 'Merge sort')
plt.plot( data['size'], data['quick sort'], label = 'Quick sort')
plt.ylabel('Running time (sec)')
plt.xlabel('Array size')
plt.legend(loc="upper left")
plt.show()
```



In the table above:

- 'size' is size of the input array.
- 'merge sort' is the actual running time of merge sort.
- 'quick sort' is the actual running time of quick sort.
- 'T(n) merge' represent the "theoretical" running time of merge sort.
- 'T(n) sort(worst)' is the "theoretical" running time of quick sort under the worst case.
- 'T(n) sort(average)' is the "theoretical" average running time of quick sort.

To better see if the actual runnint time is closely match the theoretical running time, we condsider the ratio table below which takes ratio between the actual running time and the theoretical running time for two cases. If the ratio stays at a certain number, then we can have a conclusion that the actual running time matches the theoretical running time up to a constant proportionally.

[9]:
```
ratio = {'merge' :df['merge sort']/df['T(n) merge'],
         'quick(worst)' :df['quick sort']/df['T(n) quick(worst)'],
        'quick(average)' :df['quick sort']/df['T(n) quick(average)']}
ratio = pd.DataFrame(ratio)
print(ratio)
```

```
          merge  quick(worst)  quick(average)
0           inf  9.536743e-07             inf
1  2.000643e-06  2.789497e-07    1.212825e-06
2  1.160995e-06  3.960133e-08    8.608984e-07
3  1.005348e-06  6.429195e-09    9.308231e-07
4  7.792798e-07  5.060983e-10    5.494916e-07
```

```
5  6.201979e-07  6.666191e-11    5.790182e-07
6  5.914506e-07  9.932312e-12    7.189247e-07
```

Based on the limited simulation and the results in the table above, we can see that if the input array is randomly sorted:

- for merge sort, the ratio stays around 6e-7, which means that the actual running time is captured by the theoretical running time of merge sort $O(n * log(n))$ roughly.
- for quick sort, the ratio "quick (average)" stays stable around 5~7e-7, while "quick(worst)" keeps decreasing. This means that the actual running time of the quick sort can roughly captured by the **average** theoretical runining time of quick sort $O(n * log(n))$.

1.1.2 When the inputs are already sorted

[10]:
```
import sys
# print(sys.getrecursionlimit()) ## 3000
sys.setrecursionlimit(30000)

runtime_merge2 =[]
runtime_quick2 =[]
for n in range(6):
    array = np.random.randint(low=-1e4, high=1e4, size=(10**n,))
    array.sort()
    array_c = array.copy()
    # merge sort
    st1 = time.time()
    merge_sort(array, 0, len(array) - 1)
    et1 = time.time()
    runtime_merge2.append(et1-st1)
    # quick sort
    st2 = time.time()
    quickSort(array_c, 0, len(array_c) - 1)
    et2 = time.time()
    runtime_quick2.append(et2-st2)
```

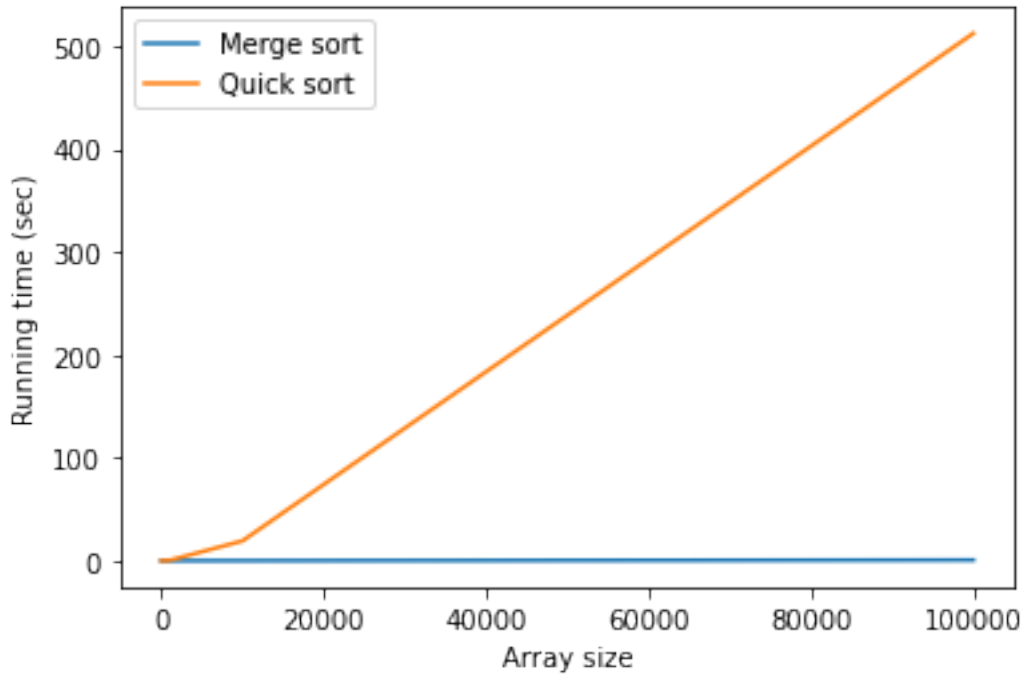
[12]:
```
size_n2 = [10**i for i in range(6)]
theory_merge2 = [math.floor(i*math.log(i)) for i in size_n2]
theory_quick_worst2 = [i*i for i in size_n2]
theory_quick_average2 = [math.floor(i*math.log(i)) for i in size_n2]

data2 = {'size': size_n2,
         'merge sort' : runtime_merge2,
        'quick sort' : runtime_quick2,
        'T(n) merge' : theory_merge2,
        'T(n) quick(worst)' : theory_quick_worst2,
        'T(n) quick(average)' : theory_quick_average2}
df2 = pd.DataFrame(data2)
print(df2)
```

```
     size  merge sort    quick sort  T(n) merge  T(n) quick(worst)  \
0       1    0.000003  9.536743e-07           0                  1
1      10    0.000035  3.600121e-05          23                100
2     100    0.000357  2.691984e-03         460              10000
3    1000    0.004812  2.643788e-01        6907            1000000
4   10000    0.060688  1.923204e+01       92103          100000000
5  100000    0.663262  5.125093e+02     1151292        10000000000

   T(n) quick(average)
0                    0
1                   23
2                  460
3                 6907
4                92103
5              1151292
```

[13]:
```
plt.plot( data2['size'], data2['merge sort'], label = 'Merge sort')
plt.plot( data2['size'], data2['quick sort'], label = 'Quick sort')
plt.ylabel('Running time (sec)')
plt.xlabel('Array size')
plt.legend(loc="upper left")
plt.show()
```



[14]:
```
ratio2 = {'merge' :df2['merge sort']/df2['T(n) merge'],
          'quick(worst)' :df2['quick sort']/df2['T(n) quick(worst)'],
```

```
        'quick(average)' :df2['quick sort']/df2['T(n) quick(average)']}
ratio2 = pd.DataFrame(ratio2)
print(ratio2)
```

```
          merge  quick(worst)  quick(average)
0           inf  9.536743e-07             inf
1  1.523806e-06  3.600121e-07        0.000002
2  7.764153e-07  2.691984e-07        0.000006
3  6.967194e-07  2.643788e-07        0.000038
4  6.589147e-07  1.923204e-07        0.000209
5  5.761022e-07  5.125093e-08        0.000445
```

Based on the limited simulation and the results in the table above, we can see that when the input array is already sorted:

- for merge sort, the ratio stays around 6e-7, which means that the actual running time is captured by the theoretical running time of merge sort $O(n * log(n))$ roughly.
- for quick sort, the ratio "quick (worst)" stays around 0.5~2e-7, while "quick(average)" keeps increasing. This means that the actual running time of the quick sort can roughly captured by the **worst case** theoretical runining time of quick sort $O(n^2)$.

1.2 The running times (best, average, and worst case) of the quickSort in O-notation

Based on the analysis provided in the lecture notes, we have the overall time complexity $T(n)$ of the quick sort to be:

$$T(n) = T(i) + T(n - i - 1) + cn,$$

where i is the number of elements in the left subarray(left of the pivot) after the partition. Depending on how the pivot is chosen, the value of i varies in each subproblem and thus affects $T(n)$.

1.2.1 Worst case

In **the worst case**, let's assume that each time the pivot is either the smallest or the largest element in the array. Then, i would be either the first or the last element of the given array. In this case, we have:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + cn + c(n-1) \\ &= \dots \\ &= cn + c(n-1) + \cdots + c \\ &= c[n(n+1)/2] \\ &= O(n^2) \end{aligned}$$

1.2.2 Best case

In the best case, suppose that each time the pivot just split the array into two parts, where the difference between two subarrays sizes is either 0 or 1. Since each subarray has $n/2$ elements at most, we have:

$T(n) = T(\frac{n-1}{2}) + T(\frac{n-1}{2} - 1) + cn \leq T(\frac{n}{2}) + T(\frac{n}{2} - 1)$ when n is odd;

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2} - 1 - 1) + cn \leq T(\frac{n}{2}) + T(\frac{n}{2} - 1)$ when n is even.

Thus:

$$T(n) \leq T(\frac{n}{2}) + T(\frac{n}{2} - 1) + cn \leq 2T(\frac{n}{2}) + cn$$

Similar to the merge sort, by the master theorem, we have $T(n) \in O(n * log(n))$

1.2.3 Average case

From the two cases above, we have an idea that the running time would be of smaller order if the partition is more balanced. We would consider the location of the pivot and see how it affects the complexity.

If we split the array evenly into 4 pieces (1/4, 1/4,1/4, 1/4), the probability that the pivot number(return by the patition() function) falls into the two middle parts is 50%. And the pivot falls into other two tails with probablity 50%.

- Suppose that the pivot number falls into the middle parts of the given array, then the worst case among all the positions would be the case that the pivot number is located at 1/4 or the 3/4 of the array. This means the array is split into 1/4 and 3/4.
- Suppose that the pivot number falls into the tails of the given array, then the worst case among all the positions would be the case that the pivot number is located at begining element(leftmost) or the end element (rightmost) of the array. This means the array is split into 0/4 and 4/4.

Suppose in the recursion tree, we have a node with cost cn.

- If we a 1/4 and 3/4 split, then we have two child nodes with $1/4cn$ cost and $3/4cn$ cost repestively, which then added up to cn cost.
- However, if we have a 0/4 and 4/4 split, two child nodes would have 0 and $c(n-1)$ cost. Since each two splits has 50% chance of happening, when there's a 1/4 and 3/4 split after the 0/4 and 4/4 split, we will have the child of the $c(n-1)$ cost node having two child nodes, with $c(n-1)/4$ and $c(n-1) * 3/4$ cost, respectively. Then the total cost is $2 * c(n-1)$.

Thus, the running time of 50% times of "1/4 and 3/4" split and 50% times of "0/4 and 4/4" split is of the same order as always having the "1/4 and 3/4" split.

Now, suppose we always have the "1/4 and 3/4" split. In the recursion tree, we would have a shortest path which consist of all "1/4" nodes and a longest path which consist of all "3/4" nodes. Since on the shortest path, each time the cost is divided by 4, we have the depth of the shortes path to be $log_4(n)$. Similary, the depth of the longest path is $log_{4/3}(n)$.

Before the shortest path reaches 1, we have the cost of all nodes at each level of the recursion tree to be cn. For example, in the second level, we have $1/4cn + 3/4cn$, etc. After the shortest path

ends, the total cost at each following levels would be less than cn. Thus, the total complexity would be at most $cn *$ the lenght of the longest path $= cn * log_{4/3}(n) = O(n * log(n))$.

Thus, given the randomness of the location of the pivot, we have the on average the runing time of the quick sort is $O(n * log(n))$.