CS/SE 2XB3 Lab 3 Report

Wang, Mingzhe
wangm235@mcmaster.ca

Li, Xing
li64@mcmaster.ca

Moon, Hyosik
moonh8@mcmaster.ca

February 5, 2021

This report concludes the main observations that we found in this week's lab, with the analysis that we did for the experiments of each methods.

1 In-Place Version

In this part, `quicksort_inplace()` is implemented in an in-place way. Experiments will be tested on this version of quicksort.

1.1 Expected advantages

Before testing, the expected advantages of `quicksort_inplace()` over `my_quicksort()` are listed below:

- **Lower Space complexity**
  Because `quicksort_inplace()` is an in-place version, i.e. it does not create copies of list, its space complexity is low. That means it can perform better in cases where the storage space is limited, e.g. cache and internal memory.

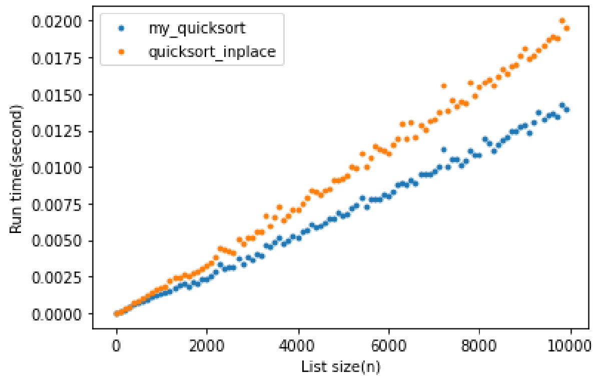
- **Possible higher performance due to no initialization**
  The version of `quicksort_copy()` actually initialize two arrays, namely `left` and `right`, each time it is called. This behavior is possible to induce longer running time. While `quicksort_recursive()` does not share the same feature – all instructions are in-place.

1.2 Test Result

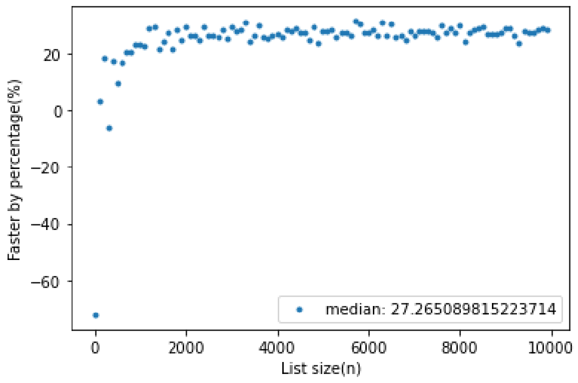
After the expected advantages are denoted, the test on the performance of `my_quicksort()` and `quicksort_inplace()` is run. All the details of this experiment can be found in the source code, while specially some of the settings are as the following:

- The method `copy()` is used to guarantee two random lists with the same elements are generated each run, instead of the wrong scenario that one list only points to another list.
- To reduce the interference from unrelated factors, for each given value of n, the test is run 10 times.
- The range of the given n is $[0, 100, 200, \ldots, 10000)$ to better present the performance of these two versions.

The results of this test are showed below.



(a) Time complexity of two versions



(b) Percentage of how much faster

Figure 1: Test results

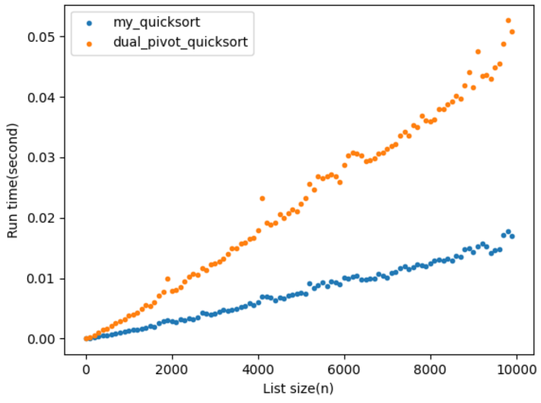
- **Which implementation is better?** From Figure 1a, it is apparent to determine that `my_quicksort()` actually performs better than `quicksort_inplace()`.
- **By how much?** From Figure 1b, the average percentage faster of `my_quicksort()` against `quicksort_inplace()` is approximately 27.265%. It should be noted that the average percentage of faster tends to be stable after $n = 1000$.
- **Which would you use in practice?** Although we decide to use `my_quicksort()` in the remainder of this lab because of its better performance, in practice, it is probably that the in-place version of quicksort will be preferred. This is because one of the advantages of quick sort is its locality, which makes quicksort usable in space limited cases like caches and internal memories. The performance loss of about 27%, can be easily compensated by the high speed of cache access.

2 Multi-Pivot

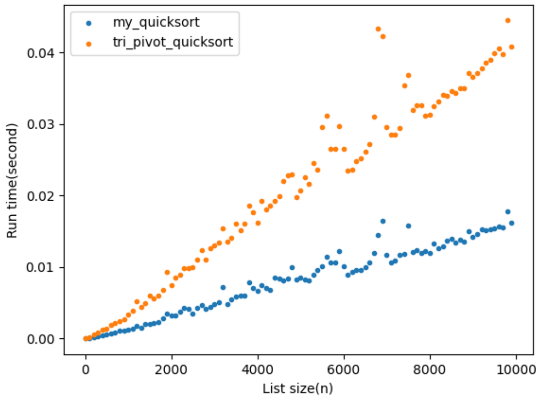
In this part, multiple pivots are used rather than single pivot in the standard quick sort algorithm. The different experiments are made against the original sort function `my_quicksort()` one by one, as requested by the lab specification. In order to compare apple-to-apple, we implement multi-pivot algorithm in the same style as `my_quicksort()`. (Figure 2a, 2b, 2c)

Based on the graph below, the algorithm with single pivot always performs better on average, compared with dual-pivot, tri-pivot or quad-pivot. Therefore, it's recommended to use single-pivot-quick-sort as default in the remainder of this lab.

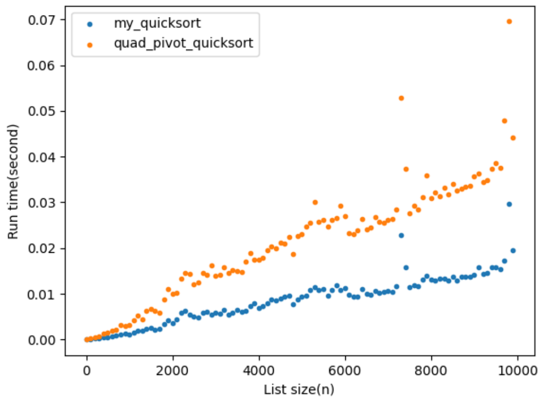
In practice, multi-pivot algorithm may have its edge when the data set is huge and has to be handled in different computers with parallel computing, such as google searching with millions of computers all over the world. In that case, multi-pivot algorithm can group data by pivots and confine further sorting within a subarray with short physical distance. The performance could be dramatically better because no cross-subarray swap is needed in each sub-array, which normally comes with high cost due to internet communication.



(a) Dual Pivot vs Single Pivot



(b) Tri Pivot vs Single Pivot



(c) Quad Pivot vs Single Pivot

Figure 2: Test results

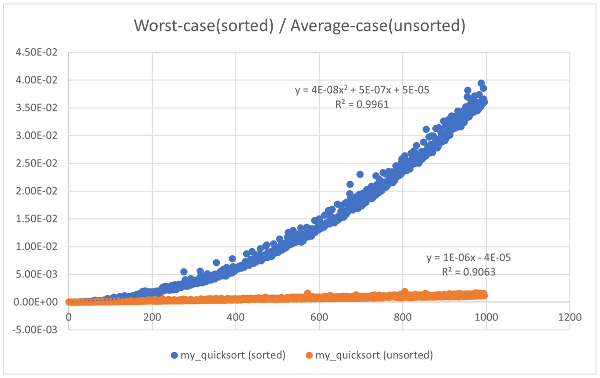
3 Worstcase Performance

In this part, tests are set to evaluate the performance of quicksort in the worst cases.

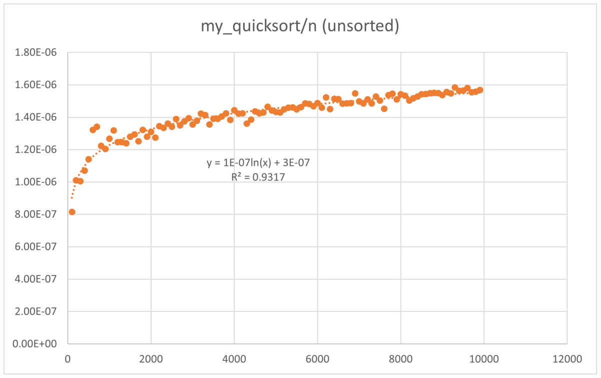
3.1 Quicksort average-case and worst-case

Quicksort's worstcase happens when the elements are already sorted and the pivot is the first or the last element. In this case, quicksort algorithm has the time complexity of

$(n-1)+(n-2)\cdots+1$, which means $\sum_{k=1}^{n-1} k \simeq O(n^2)$. On the other hand, when a list unsorted quicksort's average time complexity is $O(nlogn)$. We obtained the experimental values of the worst and average cases' time complexity (Figure 3a). In order to get the smooth graphs we ran the each point from 5 to 50 times. As a result we were able to obtain the fact that the worst-case time complexity is about $O(n^2)$, and the average-case time complexity is close to $O(n)$. In order to check more accurate time complexity of average-case, we tested it again by increasing the number of elements of the list to 10000. In the Figure 3b, we can surly see that the time complexity is about $O(nlogn)$.



(a) Worst, Average cases ($n = 1 \sim 1000$)

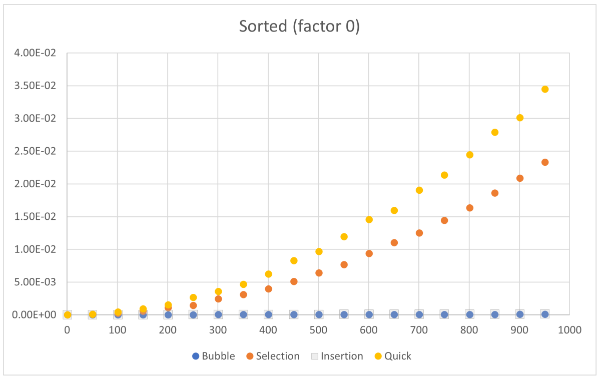


(b) Average/n ($n = 100 \sim 10000$)

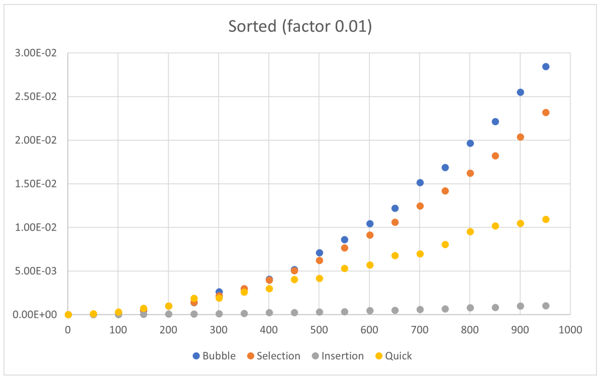
Figure 3: Time complexities of worst and average cases

3.2 Quicksort vs Elementary sorts ($n = 1 \sim 1000$, $factor = 0, 0.01$)

The second question was that among the optimized elementary sorting algorithms which one will outpeform the quicksort algorithm for a nearly sorted list. We expected that bubble and insertion will outperform the quicksort implementation, because they will stop or minimise comparison when a list is sorted. But the result was different. With the factor 0, which is totally sorted, bubble and insertion sorts surpassed the quicksort (Figure 4a). But in the case of low factor, which is 0.01, only insertion sort outperformed the quicksort (Figure 4b). This is because if there is at least one element which is not sorted in the list, bubble sort should continue to sort the list to the end. Consequently, if a list isn't sorted totally, it doesn't reduce bubble sort's time complexity. However, with insertion sort, if a side of the list is sorted, then it doesn't need to go through the remaining part of the sorted list.



(a) $factor = 0$, $n = 1 \sim 1000$

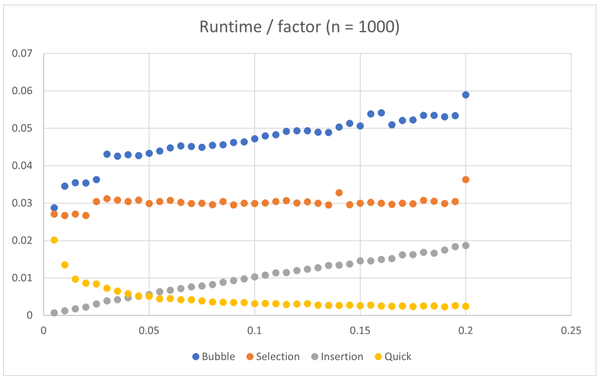


(b) $factor = 0.01$, $n = 1 \sim 1000$

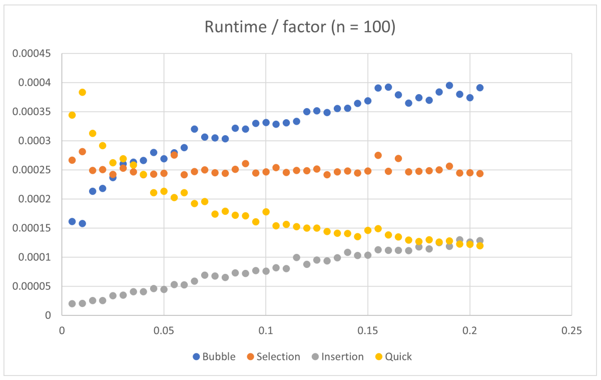
Figure 4: Time complexities of sorting algorithms

3.3 Quicksort vs Elementary sorts ($n = 1000$, $factor = 0.005 \sim 0.2$)

This section compared quicksort and elementary sorts' performance based on a list of length 1000 and near-sorted factors. The near-sorted factors change from 0.005 to 0.2, 0.005 at a time. When the value becomes 0.045, quicksort outperforms the insertion sort (Figure 5a). This result shows that if a length 1000's list is sorted more than 95.5%, then insertion sort's time performance is the best. We tested further to check that this fact can be applicable when the list's length is relatively small which is 100. As shown Figure 5b, insertion sort is still the best. This is because when a list is nearly sorted, insertion sort's time complexity is best which is close to $O(n)$, but quicksort's time complexity is worst which is close to $O(n^2)$ (Table 1).



(a) $factor = 0.005 \sim 0.2$, $n = 1000$



(b) $factor = 0.005 \sim 0.2$, $n = 100$

Figure 5: Time complexities based on sorted factors

Time complexity			
Name	**Best**	**Average**	**Worst**
Bubble sort	n	n^2	n^2
Selection sort	n^2	n^2	n^2
Insertion sort	n	$nlogn$	$nlogn$
Quicksort	$nlogn$	$nlogn$	n^2

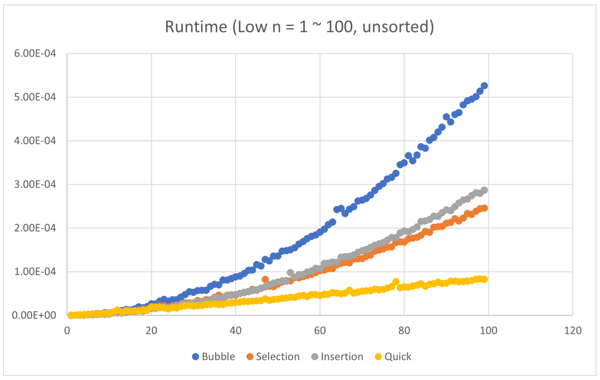
Table 1: Sorting algorithms' time complexities

4 Small Lists

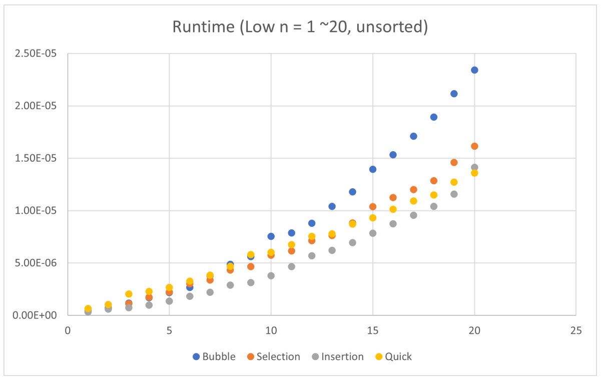
In this part, tests are set to evaluate the performance of quicksort in the cases of small lists.

4.1 Quicksort performance under small lists

In this section we tested sorting algorithms' performance under low values of n. We divided the test range into two. First n's range is 1 ∼ 100, and second range is 1 ∼ 20. And we obtained the test result by running 500 and 5000 times per point respectively for the smooth graphs. Figure 6a, 6b show that the performance of quicksort is worse than other elementary sorts under $n = 7$. When the number of elements exceeds 20, quicksort shows better performance than the others. This is because when the number of elements is small, quicksort's recursion and copy parts are dominant factors to determine the time complexity. But if the number exceeds a certain level, which is 20 in this test, the effectiveness of the algorithm begins to outperform structural shortcomings.



(a) $n = 1 \sim 100$



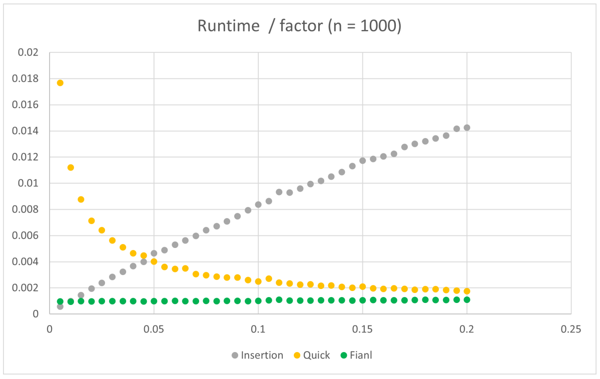
(b) $n = 1 \sim 20$

Figure 6: Time complexities on small lists

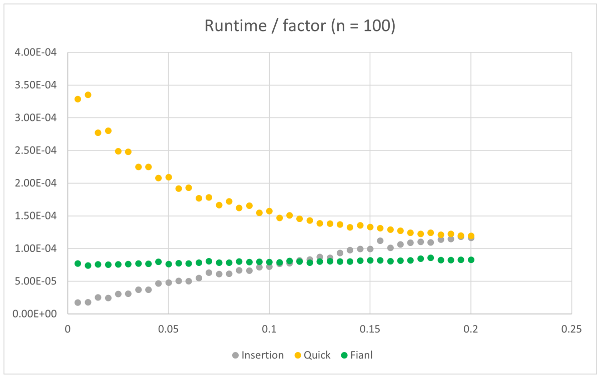
4.2 The best algorithm

We thought about the best algorithm based on the test results. In order to get the best algorithm, we considered three factors, which are '1. type of algorithms' , '2. number of elements', and '3. degree of alignment'. We consider these three factors one by one.

First, '1. type of algorithms'. In terms of time complexity, quicksort and insertion sort are the best algorithms. But there are some factors that should be considered to optimize the time complexity, which are '2. number of elements' and '3. degree of alignment'. Second, '2. number of elements'. When the number of elements are small, the difference of absoulute runtime among different sorting algorithms is trivial. Except for a specific situation, which is using the sorting algorithm with very few elements repeatedly, quicksort is the still best algorithm. If we can avoid the worst-case scenario, even though the length is small, quicksort's average time complexity is $O(nlogn)$, which means clost to n. Third, '3. degree of alignment'. When a list is nearly sorted, quicksort algorithm's effectiveness decreases significantly to $O(n^2)$. In order to avoid the worst-case scenario, we modify the `my_quicksort()` a little bit by choosing the pivot with a middle element of the list and exchanging it with the first elemnt of the list. As quicksort is recursive function, if we choose the middle element in a sorted list, we can easily avoid the smallest or largest value in the list recursively. Figure 7a, 7b are the result of `final_sort()`. Compared to the previous quicksort, they show clearly improved results in the worst case scenario.



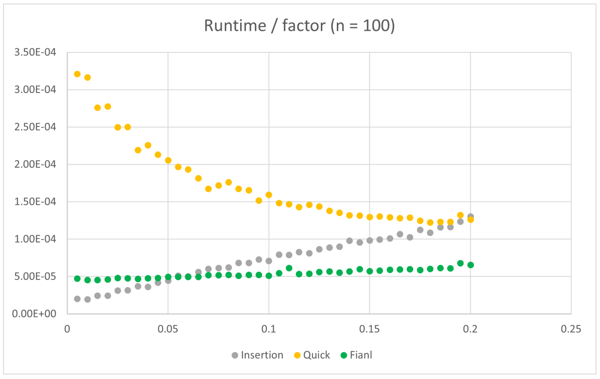
(a) $n = 1000,\ factor = 0.005 \sim 0.2$



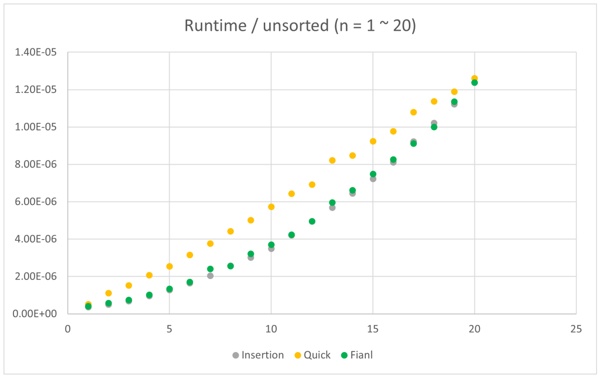
(b) $n = 100,\ factor = 0.005 \sim 0.2$

Figure 7: Time complexity of improved quicksort

In order to improve a bit more, we combined quicksort with insertion sort. Based on the test result (Figure 6b), when a list's length is less than equal 20, then we used insertion sort. As a result, `fianl_sort()`'s runtime reduced to the insertion sort's runtime level when n is samll (Figure 8a, 8b).



(a) $n = 100,\ factor = 0.005 \sim 0.2$



(b) $n = 1 \sim 20$, unsorted

Figure 8: Time complexity of final_sort()

Appendix: source code

Below is all our test code for this lab.

```
import statistics
import random
import math
import matplotlib.pyplot as plt
import timeit
from sorts import quicksort_inplace
from multi_pivot import dual_pivot_quicksort, tri_pivot_quicksort, quad_pivot_quicksort
from sorts import final_sort

### Sorting algorithms ###
def my_quicksort(L):
    copy = quicksort_copy(L)
    for i in range(len(L)):
        L[i] = copy[i]
    return L


def quicksort_copy(L):
    if len(L) < 2:
        return L
    pivot = L[0]
    left, right = [], []
    for num in L[1:]:
        if num < pivot:
            left.append(num)
        else:
            right.append(num)
    return quicksort_copy(left) + [pivot] + quicksort_copy(right)


def create_random_list(n):
    L = []
    for _ in range(n):
        L.append(random.randint(1, n))
    return L


def create_near_sorted_list(n, factor):
    L = create_random_list(n)
    L.sort()
    if factor == 0:
        return L
    for _ in range(math.ceil(n * factor)):
        index1 = random.randint(0, n - 1)
        index2 = random.randint(0, n - 1)
        L[index1], L[index2] = L[index2], L[index1]
    return L


def bubblesort_opt(L):  # Avg, Worst case bubblesort
    n = len(L)
    swap = True
    for i in range(n - 1):
        swap = False
        for j in range(n - 1 - i):
            if L[j] > L[j + 1]:
                L[j], L[j + 1] = L[j + 1], L[j]
                swap = True
        if not swap:
            break
    return L


def selectionsort(L):
    n = len(L)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if (L[j] < L[min_idx]):
                min_idx = j
        L[i], L[min_idx] = L[min_idx], L[i]
    return L
```

```
def insertionsort(L):
    n = len(L)
    for i in range(1, n):
        value = L[i]
        hole_idx = i
        while (hole_idx > 0 and L[hole_idx - 1] > value):
            L[hole_idx] = L[hole_idx - 1]
            hole_idx -= 1
        L[hole_idx] = value
    return L


### Timing experiments 1 (quicksort_inplace)###
def test_my_quicksort_vs_quicksort_inplace(runs, n):
    total_my = 0
    total_inp = 0
    for _ in range(runs):
        ls1 = create_random_list(n)
        ls2 = ls1.copy()

        start_my = timeit.default_timer()
        my_quicksort(ls1)
        end_my = timeit.default_timer()

        start_inp = timeit.default_timer()
        quicksort_inplace(ls2)
        end_inp = timeit.default_timer()

        total_my += end_my - start_my
        total_inp += end_inp - start_inp
    return (total_my / runs, total_inp / runs)


def plot_my_quicksort_and_quicksort_inplace(n_range, runs):
    t_range_my = []
    t_range_inp = []
    for n in n_range:
        t = test_my_quicksort_vs_quicksort_inplace(runs, n)
        t_range_my.append(t[0])
        t_range_inp.append(t[1])
    # plot vs.
    plt.scatter(n_range, t_range_my, marker = '.',
                label = "my_quicksort")
    plt.scatter(n_range, t_range_inp, marker = '.',
                label = "quicksort_inplace")
    plt.legend(loc='upper left')
    plt.xlabel("List size(n)")
    plt.ylabel("Run time(second)")
    plt.show()
    # plot how much
    by_how_much = []
    i = 0
    for n in n_range:
        ans = (t_range_inp[i] - t_range_my[i]) / t_range_inp[i] * 100
        by_how_much.append(ans)
        i += 1
    plt.scatter(n_range, by_how_much, marker = '.',
                label = "median: " + str(statistics.median(by_how_much)))
    plt.legend(loc='lower right')
    plt.xlabel("List size(n)")
    plt.ylabel("Faster by percentage(%)")
    plt.show()

n_range = [100 * _ for _ in range(100)]
plot_my_quicksort_and_quicksort_inplace(n_range, 10)


### Timing experiments 2 (Multi pivots) ###
def generate(f1, f2, runs, n):
    total_my = 0
    total_inp = 0
    for _ in range(runs):
        ls1 = create_random_list(n)
        ls2 = ls1.copy()

        start_my = timeit.default_timer()
        f1(ls1)
        end_my = timeit.default_timer()

        start_inp = timeit.default_timer()
        f2(ls2)
        end_inp = timeit.default_timer()

        total_my += end_my - start_my
        total_inp += end_inp - start_inp
    return (total_my / runs, total_inp / runs)


def plot(f1, f2, n_range, runs):
    t_range_my = []
    t_range_inp = []
    for n in n_range:
        t = generate(f1, f2, runs, n)
        t_range_my.append(t[0])
        t_range_inp.append(t[1])
    labels = [str(f1).split()[1], str(f2).split()[1]]
    plt.scatter(n_range, t_range_my, marker='.',
                label=labels[0])
    plt.scatter(n_range, t_range_inp, marker='.',
                label=labels[1])
    plt.xlabel("List size(n)")
    plt.ylabel("Run time(second)")
    plt.legend(loc='upper left');
    plt.savefig("Figures/multi_pivot_" + labels[0] + "_" + labels[1])
    plt.close()

# n_range = [100 * _ for _ in range(100)]
# plot(my_quicksort, dual_pivot_quicksort, n_range, 10)
# plot(my_quicksort, tri_pivot_quicksort, n_range, 10)
# plot(my_quicksort, quad_pivot_quicksort, n_range, 10)


### Timing experiments 3 (my_quicksort) ###
def timetest1(runs, n, func_type):  # unsorted
    total_runtime = 0
    for _ in range(runs):
        L_unsorted = create_random_list(n)
        # L_sorted = create_near_sorted_list(n, 0.01)
        start = timeit.default_timer()
        func_type(L_unsorted)
        end = timeit.default_timer()
        total_runtime += end - start
    return total_runtime / runs


def timetest2(runs, n, func_type, factor):  # sorted
    total_runtime = 0
    for _ in range(runs):
        # L_unsorted = create_random_list(n)
        L_sorted = create_near_sorted_list(n, factor)
        start = timeit.default_timer()
        func_type(L_sorted)
        end = timeit.default_timer()
        total_runtime += end - start
    return total_runtime / runs


## Test algorithms runtime (unsorted, low range)
for n in range(1, 21, 1):  # range(length of start_list, length of last_list, increase)
    avg_run = 10000
    print(n, #timetest1(avg_run, n, bubblesort_opt),
          #timetest1(avg_run, n, selectionsort),
          timetest1(avg_run, n, insertionsort),
          timetest1(avg_run, n, my_quicksort),
          timetest1(avg_run, n, final_sort))

## Test algorithms runtime (sorted)
# for n in range(1, 1000, 50): # range(length of start_list, length of last_list, increase)
#     avg_run = 50
#     factor = 0
#     print(n, timetest2(avg_run, n, bubblesort_opt, factor),
#           timetest2(avg_run, n, selectionsort, factor),
#           timetest2(avg_run, n, insertionsort, factor),
#           timetest2(avg_run, n, my_quicksort, factor))

## Test for different factors
# for i in range(1, 100, 1): # range(length of start_list, length of last_list, increase)
#     avg_run = 5000
#     factor = i * 0.005
#     n = 10
#      print(factor, timetest2(avg_run, n, bubblesort_opt, factor),
#            timetest2(avg_run, n, selectionsort, factor),
#            timetest2(avg_run, n, insertionsort, factor),
#            timetest2(avg_run, n, my_quicksort, factor))

## Test for different factors (Insertion, quicksort)
# for i in range(1, 41, 1): #range(length of start_list, length of last_list, increase)
#      avg_run = 1000
#      factor = i * 0.005
#      n = 100
#      print(factor, timetest2(avg_run, n, insertionsort, factor),
#            timetest2(avg_run, n, my_quicksort, factor),
#            #timetest2(avg_run, n, semi_final, factor),
#            timetest2(avg_run, n, final_sort, factor))

## Test bubblesort opt algorithms runtime (unsorted, sorted)
# for n in range(1, 1001, 1):  # range(length of start_list, length of last_list, increase)
#      avg_run = 5
#      factor = 0
#      print(n, timetest1(avg_run, n, my_quicksort),
#            timetest1(avg_run, n, quicksort_inplace),
#            timetest2(avg_run, n, bubblesort, factor),
#            timetest2(avg_run, n, bubblesort_opt, factor))

## Check sorting algorithms
# for _ in range(10):
#      L = create_random_list(20)
#      # print(L, bubblesort_opt(L.copy()))
#      # print(L, selectionsort(L.copy()))
#      print(L, insertionsort(L.copy()))
#      print(L, my_quicksort(L.copy()))
#      # print(L, semi_final(L.copy()))
#      print(L, final_sort(L.copy()))
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Feb  1 21:47:40 2021
@author: Mingzhe Wang, Hyosik Moon, Xing Li
"""

#@@ brief This function inplace quicksort
#@  param L The list to be sorted
#@  param low First index of a list
#@  param high Last index of a list
#@  return partition index
# note: Always pick first element as pivot. (We want to control vairables in
#       our experiment, so it it better to keep consistent with my_quicksort).
def quicksort_inplace(L):
    quicksort_recursive(L, 0, len(L) - 1)

def quicksort_recursive(L, low, high):
    if low < high:
        par = partition(L, low, high)
        quicksort_recursive(L, low, par - 1)
        quicksort_recursive(L, par + 1, high)

def partition(L, low, high):
    pivot = L[low]
    i = high + 1
    for j in range(high, low - 1, -1):
        if L[j] > pivot:
            i -= 1
            L[i], L[j] = L[j], L[i] #swap
    L[i - 1], L[low] = L[low], L[i - 1] #swap
    return i - 1


#@@ brief This function recursively run quicksort with multiple pivots
#@  param L The list to be sorted
#@  param n The number of pivots
#@  return partially sorted array with multiple pivots
def quicksort_copy(L,n):
    if len(L) < 2:
        return L
    pivot_num = min(n, len(L)-1) # Adjust when the length is too short
    pivots = quicksort_copy(L[0:pivot_num], 1) # Set up a list of pivots
    containers = [[] for _ in range(pivot_num)] # Set up a list of containers for between pivots
    containers.append([]) # The number of containers is always one more than the number of pivots
    for num in L[pivot_num:]: # For all elements other than pivots
        for i in range(pivot_num): # Compare an element with each pivot and put in one container
            if num < pivots[i]:
                containers[i].append(num)
                break
        if num >= pivots[pivot_num - 1]:
            containers[pivot_num].append(num)
    result = []
    for i in range(pivot_num): #concatenate the elements in each container as well as pivots
        result += quicksort_copy(containers[i],n) + [pivots[i]]
    return result + quicksort_copy(containers[pivot_num],n)


def dual_pivot_quicksort(L):
    copy = quicksort_copy(L, 2)
    for i in range(len(L)):
        L[i] = copy[i]


def tri_pivot_quicksort(L):
    copy = quicksort_copy(L, 3)
    for i in range(len(L)):
        L[i] = copy[i]


def quad_pivot_quicksort(L):
    copy = quicksort_copy(L, 4)
    for i in range(len(L)):
        L[i] = copy[i]


#@@ brief This function is the optimized quicksort
#@  param L The list to be sorted
#@  return sorted list
def final_sort(L):
    if len(L) <= 20:
        return insertionsort(L)
    copy = final_sort_copy(L)
    for i in range(len(L)):
        L[i] = copy[i]
    return L


def final_sort_copy(L):
    if len(L) <= 20:
        return insertionsort(L)
    mid = len(L)//2
    pivot = L[mid]
    L[0], L[mid] = L[mid], L[0]
    left, right = [], []
    for num in L[1:]:
        if num < pivot:
            left.append(num)
        else:
            right.append(num)
    return final_sort_copy(left) + [pivot] + final_sort_copy(right)


def insertionsort(L):
    n = len(L)
    for i in range(1, n):
        value = L[i]
        hole_idx = i
        while (hole_idx > 0 and L[hole_idx - 1] > value):
            L[hole_idx] = L[hole_idx - 1]
            hole_idx -= 1
        L[hole_idx] = value
    return L
```