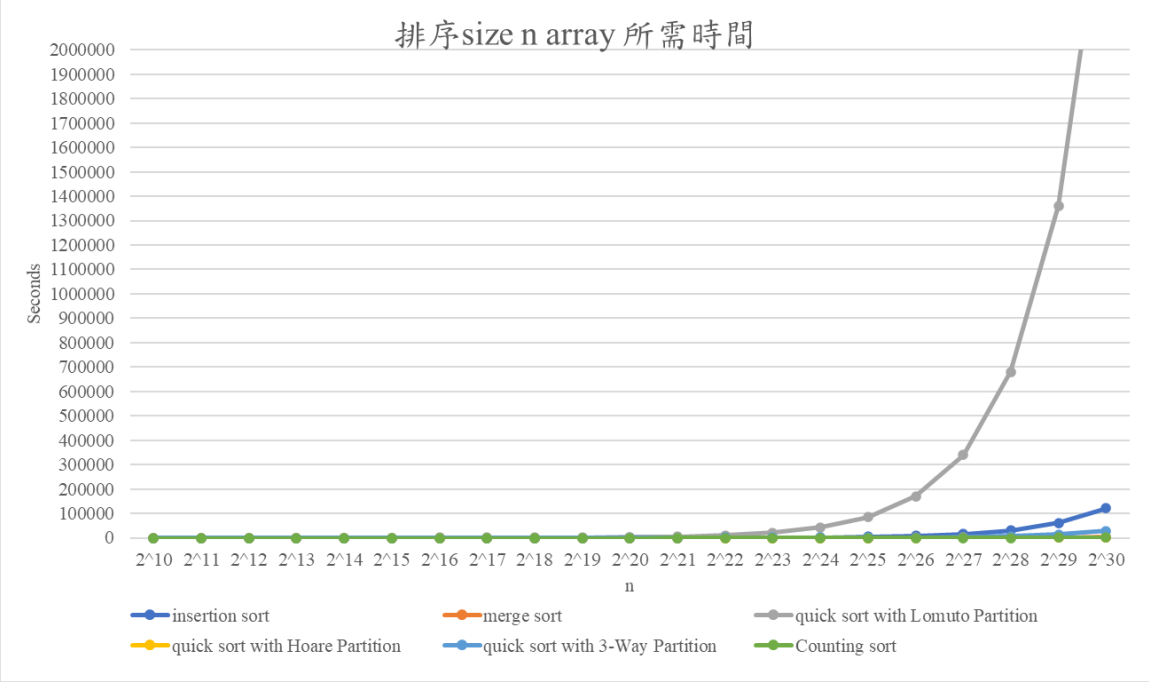
實作目標：比較insertion sort, merge sort, 三種randomized quick sort (Lomuto Partition, Hoare Partition, 與3-Way Partition)與Counting sort。

一、 三張折線圖與解釋實驗結果（每張圖6條線，對應到上述6個演算法的執行時間）

甲、(20%)圖一產生方法：

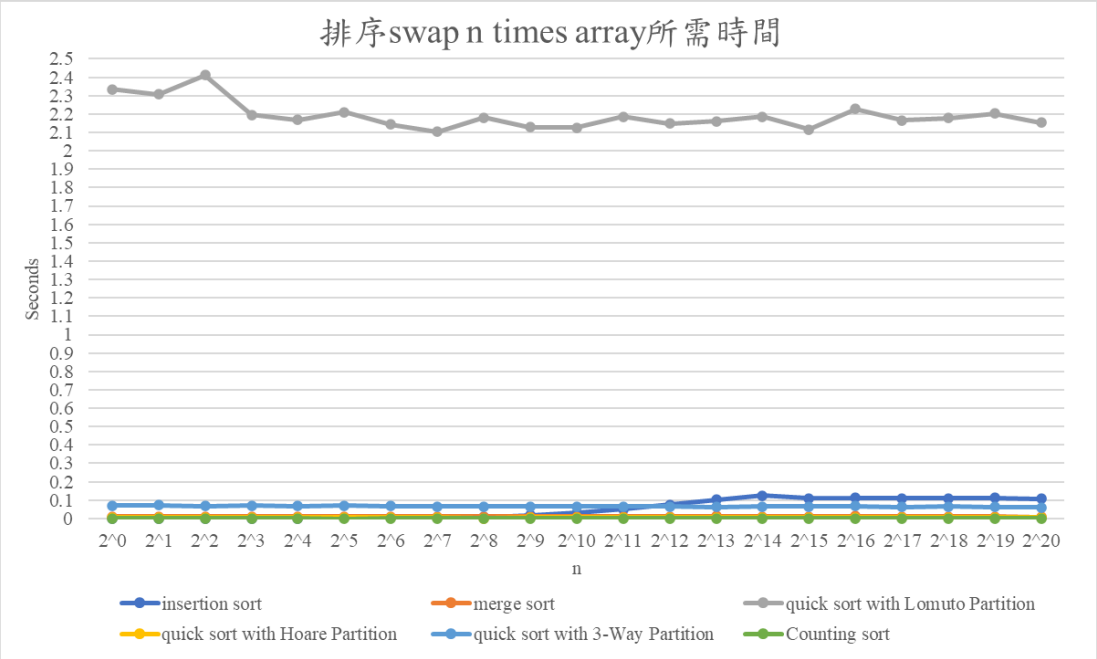
i. 產生一個大小為n的array，$n = 2^{10}, 2^{11}, \dots, 2^{30}$。
ii. 每個arr[i]為$0 \sim n-1$隨機產生。
iii. x軸為n值：$n = 2^{10}, 2^{11}, \dots, 2^{30}$。
iv. y軸為每個演算法的執行時間（產生10個input後取平均）。



- 隨array變大，quick sort with Lomuto partition排序時間增加最快，insertion sort次之。
- Insertion sort在array size小 (幾乎排序好) 時，排序速度和其他演算法差不多，在size變大時才較慢。

乙、(20%)圖二產生方法：

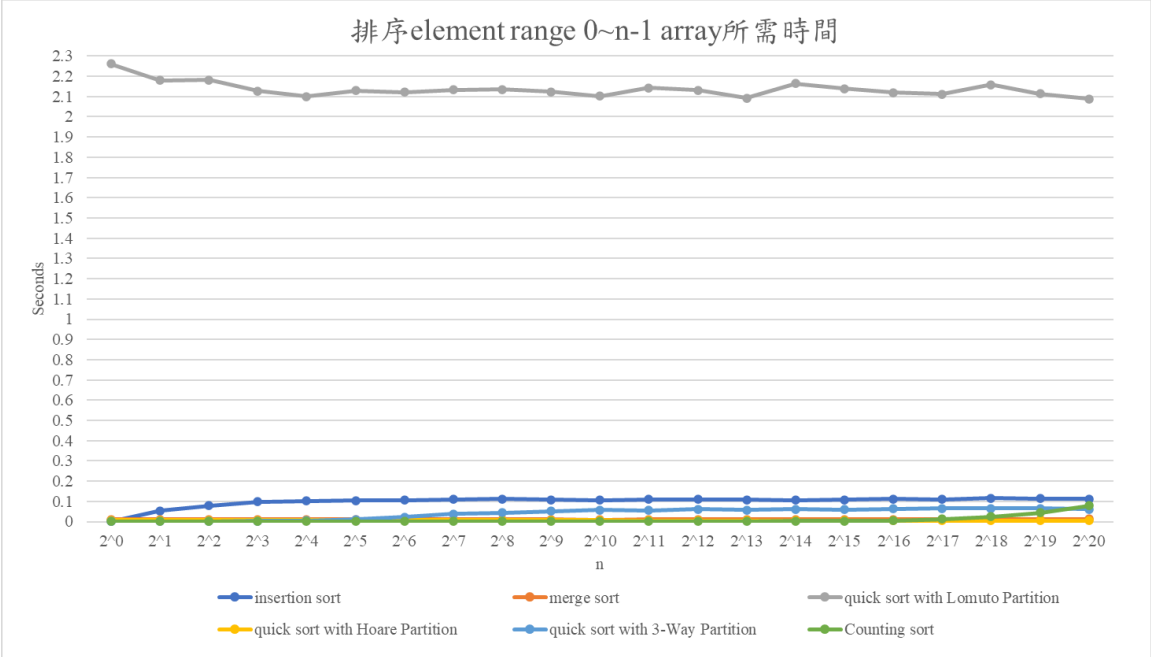
i. 產生一個大小為2^{20}的排序好的array（換句話說，arr[i]=i）。
ii. 隨機swap array內容k次，$k = 2^0, 2^1, 2^2, \dots, 2^{20}$。
iii. x軸為k值：$k = 2^0, 2^1, 2^2, \dots, 2^{20}$。
iv. y軸為每個演算法的執行時間（產生10個input後取平均）。



- 因電腦效能，陣列大小調整為2^13。
- 假說一：insertion sort是O(n^2)，但在幾乎排序好的array(k小)時，排序的速度比merge sort、quick sort快，隨著array變亂(k增加)，insertion sort才會變得比較慢。quick sort 若用 Lomuto partition是O(n^2)和insertion sort可能差不多慢。
- 實驗結果驗證假說，但是Lomuto partition比insertion sort慢得多，可能因為我選擇的Lomuto寫法是把array最右邊當作pivot，不是切在中間，使得recursive次數增加，執行時間變長。

丙、(20%)圖三產生方法：

i. 產生一個大小為2^{20}的array。
ii. 每個arr[i]為$0 \sim k-1$隨機產生。
iii. x軸為k值：$k = 2^0, 2^1, 2^2, \ldots, 2^{20}$。
iv. y軸為每個演算法的執行時間（產生10個input後取平均）。



- 因電腦效能，陣列大小調整為2^13。
- 假說一：k小，array中元素相同的很多，反之，k大，array中元素大多不同。Quick sort不同partition方法下，當元素大多相同時，執行時間三種方法差比較多， 元素不同時，三種方法時間差不多。
- 實驗結果驗證假說一。
- 假說二：k小，執行時間：Lomuto > Hoare > 3-Way。
- 實驗結果和假說二不同，Hoare比3-Way快，可能因為pivot位置選在中間。

二、心得、疑問、與遇到的困難。

- 可改善：比較 quick sort 的三種 partition 時，可能需要 pivot 的切法都切在中間或是都切在極端，才可以互相比較。此實作，Lomuto 因為需要把 pivot 和最後一個數對調，所以通常把最後一個數字當作 pivot，3-Way 把第一個數字當 pivot，Hoare 中位數作 pivot，可調整 Lomuto 、3-Way、Hoare 都選擇中位數的 pivot 比較基準較一致。

- 可改善：第 2 種 input array size = 2**13，可能造成 swap 次數沒有因為越大而越排越亂，比如說 swap2**20 次，因為早就大於 array size，所以弄亂的 element 可能又被 swap 回去。

三、程式碼與註解，請特別註明 Lomuto partition 中 swap pivot 與 array 最後一個 element 的 code。

```python
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

```python
def countingSort(input_array):
    # Finding the maximum element of input_array.
    M = max(input_array)

    # Initializing count_array with 0
    count_array = [0] * (max(input_array) + 1)

    # Mapping each element of input_array as an index of count_array
    for num in input_array:
        count_array[num] += 1

    # Calculating prefix sum at every index of count_array
    for i in range(1, max(input_array) + 1):
        count_array[i] += count_array[i - 1]


    # Creating output_array from count_array
    output_array = [0] * len(input_array)

    for i in range(len(input_array) - 1, -1, -1):
        output_array[count_array[input_array[i]] - 1] = input_array[i]
        count_array[input_array[i]] -= 1
    return output_array
```

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)  # 切到LR都只剩一個元素

        i = j = k = 0
        # Copy data to new arrays #從L[0]R[0]開始比，小的放進arr，index++
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):  # Checking if any element was left #比完剩下的放進arr
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

```python
def partition(arr, low, high):
    pivot = arr[high] #pivot is the last element in the array
    i = (low - 1)
    for j in range(low, high):
        # If current element is smaller than or equal to pivot
        if (arr[j] <= pivot):
            # increment index of smaller element
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)


''' The main function that implements QuickSort
arr --> Array to be sorted,
low --> Starting index,
high --> Ending index '''
def quickSortLomuto(arr, low, high):
    while (low < high):
        ''' pi is partitioning index, arr[p] is now at right place '''
        pi = partition(arr, low, high)
        if pi - low < high - pi:
            quickSortLomuto(arr, low, pi - 1)
            low = pi + 1
        else:
            quickSortLomuto(arr, pi + 1, high)
            high = pi - 1
```

```python
def partition(arr, low, high):
    mid = (low + high) // 2  # use the middle element as pivot
    pivot = arr[mid]
    i = low - 1
    j = high + 1
    while (True):
        i += 1
        while (arr[i] < pivot):
            i += 1
        j -= 1
        while (arr[j] > pivot):
            j -= 1
        # If two pointers met.
        if (i >= j):
            return j
        arr[i], arr[j] = arr[j], arr[i]

def quickSortHoare(arr, low, high):
    ''' pi is partitioning index, arr[p] is now at right place '''
    while (low < high):
        pi = partition(arr, low, high)
        if pi - low < high - pi:
            quickSortHoare(arr, low, pi)
            low = pi + 1
        else:
            quickSortHoare(arr, pi + 1, high)
            high = pi
```

```python
def quickSort3way(arr, low, high):
    if low < high:
        pivot = arr[low]
        left, right, i = low, high, low + 1

        while i <= right:
            if arr[i] < pivot:
                arr[i], arr[left] = arr[left], arr[i]
                i += 1
                left += 1
            elif arr[i] > pivot:
                arr[i], arr[right] = arr[right], arr[i]
                right -= 1
            else:
                i += 1

        quickSort3way(arr, low, left - 1)
        quickSort3way(arr, right + 1, high)
```

```python
import random
import timeit
from insertionSort import insertionSort
from mergeSort import mergeSort
from quickSortLomuto import quickSortLomuto
from quickSortHoare import quickSortHoare
from quickSort3way import quickSort3way
from countingSort import countingSort
import sys
import cProfile

def runSortingAlgorithm(algorithm, arr):
    return timeit.timeit(lambda: algorithm(arr), number=numTrials) / numTrials

def generateArraySize(size):
    return [random.randint(0, size-1) for s in range(size)]

def generateArrayElementRange(k):
    return [random.randint(0, k-1) for s in range(2**13)]

def random_swap(arr, swap):
    for s in range(swap):
        i, j = random.sample(range(len(arr)), 2)
        arr[i], arr[j] = arr[j], arr[i]
        # print(f"Swap {s+1}:", arr)
    return arr
```

```python
'''To compare algorithms, the following are 3 kinds of output and
we can use each output to run the main separately.'''

if __name__ == '__main__':
    numTrials = 10
    # First input
    sizes = [2**i for i in range(10, 31)]
    for size in sizes:
        arr = generateArraySize(size)

    #The second input
    swaps = [2**i for i in range(0, 21)]
    for swap in swaps:
        sortedArray = list(range(2**13))
        arr = random_swap(sortedArray, swap)
        #print(f"Original array (swap times {swap}):", arr)

    # The third input
    ks = [2**i for i in range(0, 21)]
    for k in ks:
        arr = generateArrayElementRange(k)
        #print(f"Original array (element range 0~{k}):", arr)

        averageInsertionSortTime = runSortingAlgorithm(insertionSort, arr.copy())
        averageMergeSortTime = runSortingAlgorithm(mergeSort, arr.copy())
        averageQuickSortLomutoTime = runSortingAlgorithm(lambda arr: quickSortLomuto(arr, 0, len(arr) - 1), arr.copy())
        averageQuickSortHoareTime = runSortingAlgorithm(lambda arr: quickSortHoare(arr, 0, len(arr) - 1), arr.copy())
        averageQuickSort3wayTime = runSortingAlgorithm(lambda arr: quickSort3way(arr, 0, len(arr) - 1), arr.copy())
        averageCountingSortTime = runSortingAlgorithm(countingSort, arr.copy())
```

```python
print(f"size{size}, {averageInsertionSortTime}, {averageMergeSortTime}, {averageQuickSortLomutoTime}, {averageQuickSortHoareTime}, {averageQuickSort3wayTime}, {averageCountingSortTime}")
print(f"swap times {swap}, {averageInsertionSortTime}, {averageMergeSortTime}, {averageQuickSortLomutoTime}, {averageQuickSortHoareTime}, {averageQuickSort3wayTime}, {averageCountingSortTime}")
print(f"element range 0~{k}, {averageInsertionSortTime}, {averageMergeSortTime}, {averageQuickSortLomutoTime}, {averageQuickSortHoareTime}, {averageQuickSort3wayTime}, {averageCountingSortTime}")
```