DS_HW2_資訊三_嚴聲遠_111703009

程式碼運作邏輯設計：

利用9個不同的腳本(.py)執行分別的排序演算法，9個腳本分別負責跑出九張折線圖。

詳細完整程式碼、成果請至：https://github.com/spaces-lalala/2024DS_HW/tree/Hw1/hw1

排序程式詳細說明：

1. **Insertion sort**：
   a. 算法遍歷每個元素，將它插入到之前已排序的部分的適當位置，從而逐步建立已排序的數列。
   b. 每次內層 while 循環會將 key 與前面的元素(從 arr[j]，也就是以排序好的陣列開始往前看)比較，直到找到正確的位置，並將 key 插入到這個位置。

```
void insertion_sort(std::vector<int>& arr) {
    for (size_t i = 1; i < arr.size(); ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

2. **Merge sort**：
   a. Merge sort是一種Divide-and-conquer排序算法。它將Array不斷地分成較小的子Array，直到每個子Array只有一個元素，然後將這些子Array合併成一個有序數組。
   b. merge_sort 函數：遞迴的將Array分為左右兩半，直到每個子Array的大小為 1，然後使用 merge 函數將這些子Array合併。

c. merge 函數：將兩個已排序的子Array拷貝到臨時陣列中。比較這兩個子Array的元素，將較小的依次放回原陣列，最後將剩餘的元素全部拷貝回原Array，完成合併。

```cpp
void merge_sort(std::vector<int>& arr, int l, int r) {
    if (l >= r) return;
    int m = l + (r - l) / 2;
    merge_sort(arr, l, m);
    merge_sort(arr, m + 1, r);
    merge(arr, l, m, r);
}
```

```cpp
void merge(std::vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    std::vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; ++i) L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

3. Randomized quick sort(**Lomuto、Hoare、3way**)：

   a. Quicksort 函數(圖片以hoare舉例，另兩個方式僅套用不同p函數)：

      A. Partitioning(p)：QuickSort 使用一個「pivot」將陣列分成兩個部分(3way 則多一個部分)，左邊的元素都比樞軸小，右邊的元素都比樞軸大。

      B. Recursion：對分割後的兩部分再次進行 QuickSort，直到陣列變得足夠小。

   b. with **Lomuto Partition**：

      A. 選擇Pivot：隨機選擇一個Pivot並將其移至陣列的結尾位置。

      B. Partition邏輯：使用指標 i 來找小於或等於Pivot的元素範圍。遍歷陣列並將小於或等於Pivot的元素與 i 所指向的元素交換。最後，將樞軸移到正確位置（i + 1）。Return Pivot的位置 i + 1 作為分割點。

```cpp
int lomuto_partition(std::vector<int>& arr, int low, int high) {
    int random_index = low + rand() % (high - low + 1);
    std::swap(arr[random_index], arr[high]);
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

c. with **Hoare Partition**：

A. 選擇Pivot：隨機選擇一個Pivot並將其移至陣列的Head位置。

B. Partition邏輯：使用兩個指標 i 和 j，i 從左到右掃描，j 從右到左掃描。當 arr[i] 大於或等於Pivot，且 arr[j] 小於或等於pivot時，交換兩個元素。當 i 與 j 相遇時，返回 j 作為分割點。

```
int hoare_partition(std::vector<int>& arr, int low, int high) {
    int random_index = low + rand() % (high - low + 1);
    std::swap(arr[low], arr[random_index]);
    int pivot = arr[low];
    int i = low - 1, j = high + 1;
    while (true) {
        do { i++; } while (arr[i] < pivot);
        do { j--; } while (arr[j] > pivot);
        if (i >= j) return j;
        std::swap(arr[i], arr[j]);

    }
}
```

d. with **3way Partition**：

A. 選擇Pivot：隨機選擇一個Pivot並將其移至陣列的結尾位置。

B. Partition邏輯：使用指標 i 來找小於pivot的元素，j 追踪大於pivot的元素。當 arr[i] 等於pivot時，將其移動到中間部分。將陣列分成三部分：小於樞軸、等於樞軸、大於樞軸。返回 i 和 j 以進一步分割。

```
void way3_partition(std::vector<int>& arr, int l, int r, int &i, int &j) {
    int random_index = l + rand() % (r - l + 1);
    std::swap(arr[random_index], arr[r]);
    i = l - 1, j = r;
    int p = l - 1, q = r;
    int v = arr[r];
    while (true) {
        while (arr[++i] < v);
        while (v < arr[--j]) if (j == l) break;
        if (i >= j)break;
        swap(arr[i], arr[j]);
        if (arr[i] == v) {
            p++;
            swap(arr[p], arr[i]);
        }
        if (arr[j] == v) {
            q--;
            swap(arr[j], arr[q]);
        }
    }
    swap(arr[i], arr[r]);
    j = i - 1;
    for (int k = l; k < p; k++, j--)
        swap(arr[k], arr[j]);
    i = i + 1;
    for (int k = r - 1; k > q; k--, i++)
        swap(arr[i], arr[k]);
}
```

4. **Counting sort**：

   a. 尋找最大值：首先找出陣列中的最大值 max_val，用來設定計數陣列的大小。

   b. 計數陣列初始化：建立一個大小為 max_val + 1 的計數陣列 count，並將其元素全部初始化為 0。

   c. 計數每個元素的出現次數：遍歷原陣列 arr，對應的數值在計數陣列中遞增。

   d. 重建排序後的陣列：使用計數陣列來重新填充原始陣列 arr，使其排序完成。

```
void counting_sort(std::vector<int> &arr) {
    int max_val = *std::max_element(arr.begin(), arr.end());
    std::vector<int> count(max_val + 1, 0);
    for (int x : arr)
        count[x]++;
    int index = 0;
    for (int i = 0; i <= max_val; ++i) {
        while (count[i]--)
            arr[index++] = i;
    }
}
```

5. **Quick_merge_sort_s & Merge_quick_sort_s**：

   a. **Quick_merge_sort_s**：

      A. 使用已於上面介紹過的函數(包含hoare_Partition、merge、merge_sort)

      B. quick_merge_sort為主體：

```
void quick_merge_sort(std::vector<int>& arr, int low, int high, int s) {
    if (high - low + 1 <= s) {
        merge_sort(arr, low, high);
    } else if (low < high) {
        int p = hoare_partition(arr, low, high);
        quick_merge_sort(arr, low, p, s);
        quick_merge_sort(arr, p + 1, high, s);
    }
}
```

C. 陣列大小判斷：QuickSort (Hoare Partition)進行分割，將陣列分成更小的部分。如果子陣列的大小 (high - low + 1) 小於或等於 s，則使用 Merge Sort 進行排序。

D. 遞迴的對兩部分使用 quick_merge_sort 進行處理，直到每個子陣列足夠小(以s判斷)，轉而使用 Merge Sort 來完成排序。

b. **Merge_quick_sort_s**：

A. 使用已於上面介紹過的函數(包含hoare_Partition、merge、quick_sort)

B. merge_quick_sort為主體：

```
void merge_quick_sort(std::vector<int>& arr, int left, int right, int s) {
    if (right - left + 1 <= s) {
        quick_sort(arr, left, right);
    } else if (left < right) {
        int mid = left + (right - left) / 2;
        merge_quick_sort(arr, left, mid, s);
        merge_quick_sort(arr, mid + 1, right, s);
        merge(arr, left, mid, right);
    }
}
```

C. 陣列大小判斷：Merge_sort進行分割，將陣列分成更小的部分。如果子陣列的大小 (right - left + 1) 小於或等於 s，則使用 QuickSort 進行排序

D. 遞迴的對兩部分使用 merge_quick_sort 進行處理，直到每個子陣列足夠小(以s判斷)，轉而使用 Quick Sort(Hoare) 來完成排序。

6. 主函式(Main)：在所有的 main() 函式中，程序都通過命令列參數來接收測試的資料量 n&k 和運行模式 mode。不同的 mode 決定了程式將如何進行效率測試。分別做出折線圖並輸出。

7. 腳本：這些腳本的核心目的是透過執行 C++ 程式來分析不同排序演算法的表現。每個腳本都處理不同的測試情境，並且使用 Python 來管理執行過程、處理結果、進行線性回歸補充缺失資料，最後將數據視覺化跑出折線圖。
8. 預測：

   利用線性回歸來填補因測試超時而缺失的資料點。取現有數據後使用 scikit-learn 的 LinearRegression 模型進行回歸訓練。

```python
def predict_missing_points(csv_file):
    # 讀取資料
    data = pd.read_csv(csv_file)

    # 確保有足夠的資料來做預測
    if len(data) < 2:
        return None

    # 取出 n 和 time 欄位
    X = data['size'].values.reshape(-1, 1)
    y = data['time'].values

    # 使用線性回歸來預測
    model = LinearRegression()
    model.fit(X, y)

    return model
```

```python
# 預測缺失資料點並將預測值加入 CSV
for name, output_file in output_files.items():
    model = predict_missing_points(output_file)
    if model and missing_points[name]["size"]:
        for size in missing_points[name]["size"]:
            predicted_time = model.predict(np.array([[size]]))[0]
            with open(output_file, 'a') as f:
                f.write(f"{size},{predicted_time}\n")
            missing_points[name]["predicted_time"].append(predicted_time)
```

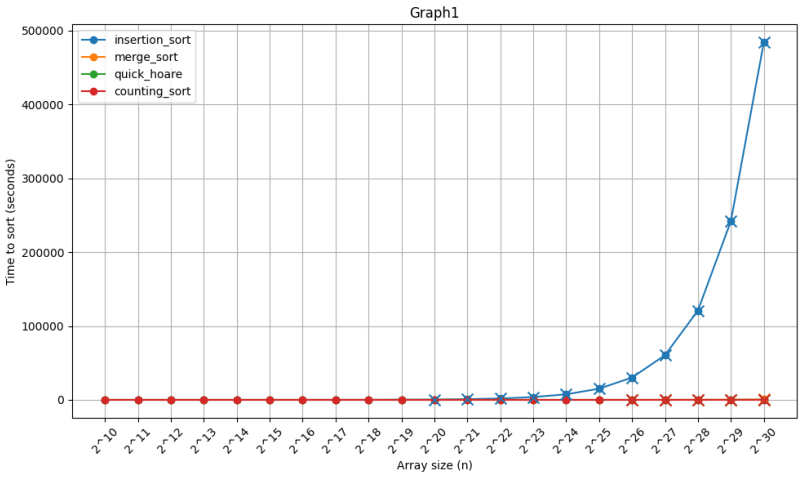
實驗圖分析：

X 為預測資料點。

詳細折線圖與數據同前文連結：https://github.com/spaces-alala/2024DS_HW/tree/Hw1/hw1

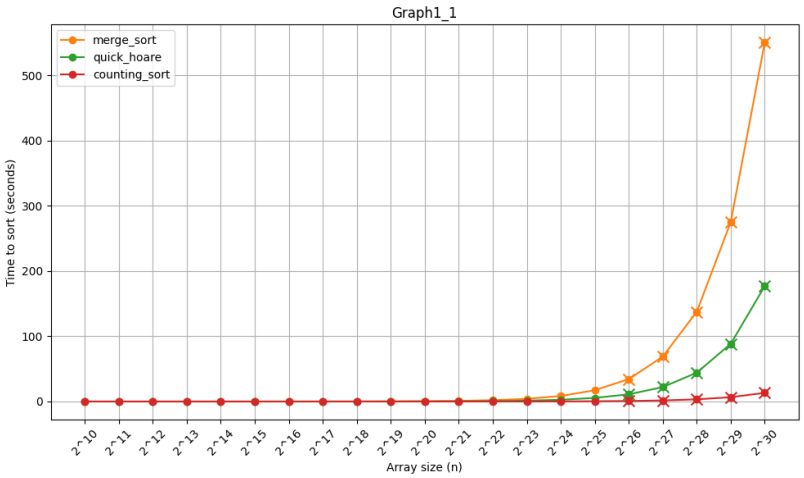
1. 第一張折線圖：

折線圖：



觀察 csv 後發現，由於 insertion_sort 數值過大，壓縮了其他資料結構的顯示，所以我做了一個去除 insertion_sort 的版本。

折線圖(without insertion)：

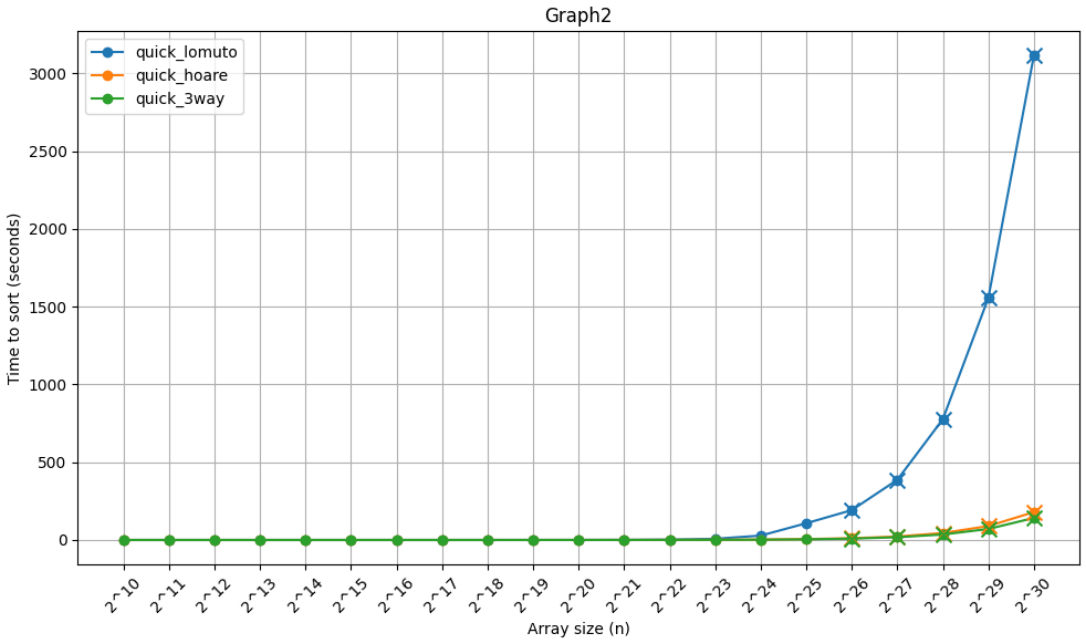


分析：

由於 insertion sort 的時間複雜度是 $O(n^2)$，對於大型資料，它的執行時間遠遠超過其他排序演算法。因此，當數據規模較大時(圖中可發現 225 後可發現急速上升)，insertionsort 的時間急劇上升。在 1_1 圖中可發現 Quick sort 比 Merge sort 在資料量大時，所需時間會相對較少。由於 Merge Sort 會導致大量非連續的記憶體訪問，相對 Quick sort 會有較多的 Cache miss。導致

Merge sort 需要更多時間。而 Counting sort 的時間複雜度是 O(n+k)，因此即使資料量增加，其速度也遠快於其他基於比較的排序演算法。

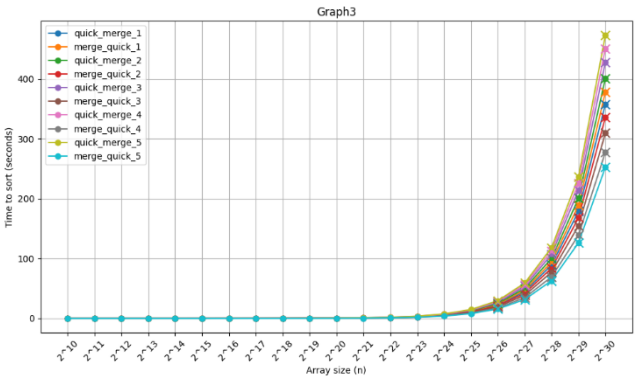
2. 第二張折線圖：



分析：

Lomuto 的效率明顯低於另外兩種 Partition，那是由於 Lomuto 每次 arr[j] 小於等於 pivot 時，都需要進行交換操作。即使 arr[i] 和 arr[j] 可能指向同一個元素（當 i 與 j 相等時），這些不必要的交換操作仍然會被執行。相對於 Hoare 分割法來說，Lomuto 分割法的交換次數明顯更多。3way Partition 及 Hoare Partition 則相對穩定且高效，並不會有過多多餘的 swap。

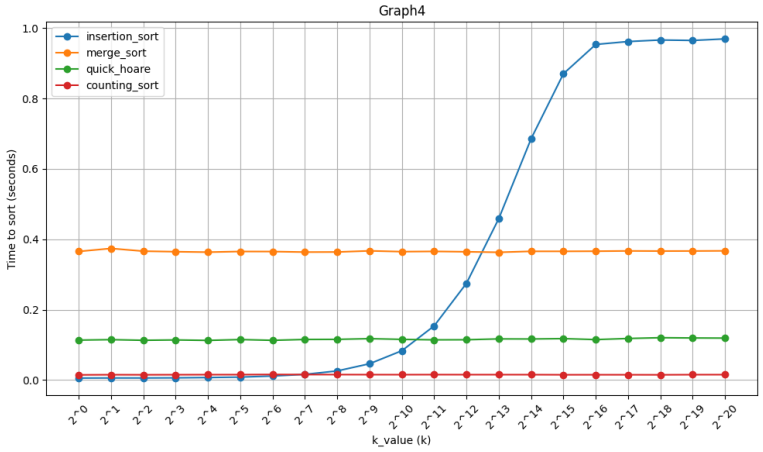
3. 第三張折線圖：



分析：

Quick Sort 的平均時間複雜度是 O(nlogn)，而 Merge Sort 雖然理論上也有 O(nlogn)的時間複雜度，但實際上，Quick Sort 在許多情況下會比 Merge Sort 快(從第一張圖分析也可得知)，它在內存使用上更加有效（不需要額外的儲存空間來合併子陣列。因此，當主要使用 Quick Sort 處理較大區段時，它可以更快地完成排序，也因此會有大部分 Quick_merge 較 Merge_quick 快。

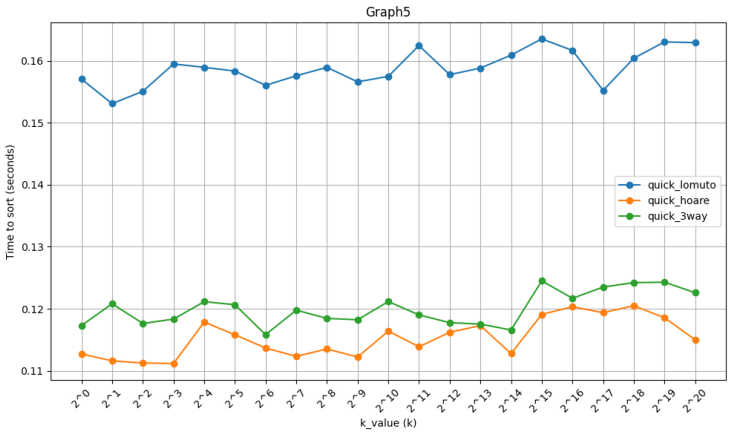
4. 第四張折線圖：



分析：

Insertion sort 在幾乎已排序的資料上表現良好(O(n))，但對於完全隨機的資料，時間複雜度為 O(n2)。故在 k 低於 213 值（即少量交換）時，陣列接近排序好的狀態，插入排序速度較快。但隨著 k 增加，陣列逐漸變得隨機，插入排序的時間急劇增加。由於我最一開始建立排序好的陣列非為 Value 皆相同的情況，而對於 Merge sort(無論 Input Array 皆為 O(nlogn))、Quick sort(取決於 pivot 選擇，但此次皆為 Randomized pivot)、Counting sort(無論 Input Array 皆為 O(n+k))並不會因為陣列是否接近排序好的陣列，而大幅影響到執行所需時間。此三種的所需時間大小關係則如同圖 1 中分析的為 Counting sort 最快，最慢的是 Merge sort。

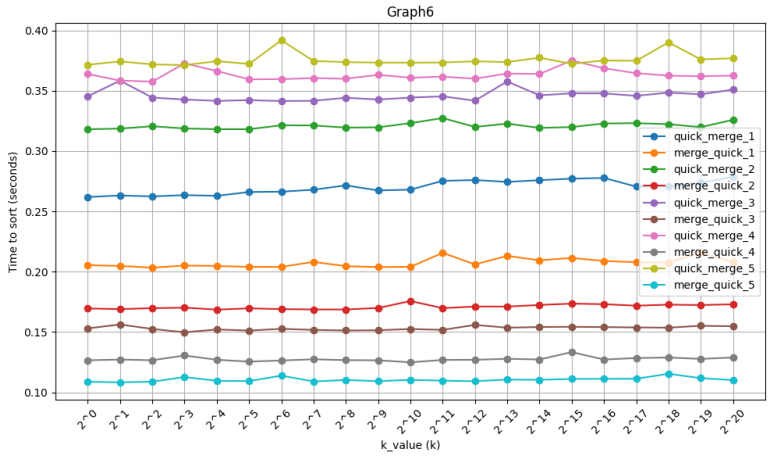
5. 第五張折線圖：



分析：

此圖符合於第二張與第四張圖中分析的綜合推論，如圖四中提及 Quick sort 取決於 pivot 選擇，但此次皆為 Randomized pivot，並不會因為陣列是否接近排序好的陣列，而大幅影響到執行所需時間，而圖二解釋了 Lomuto 相對時間較多的原因。

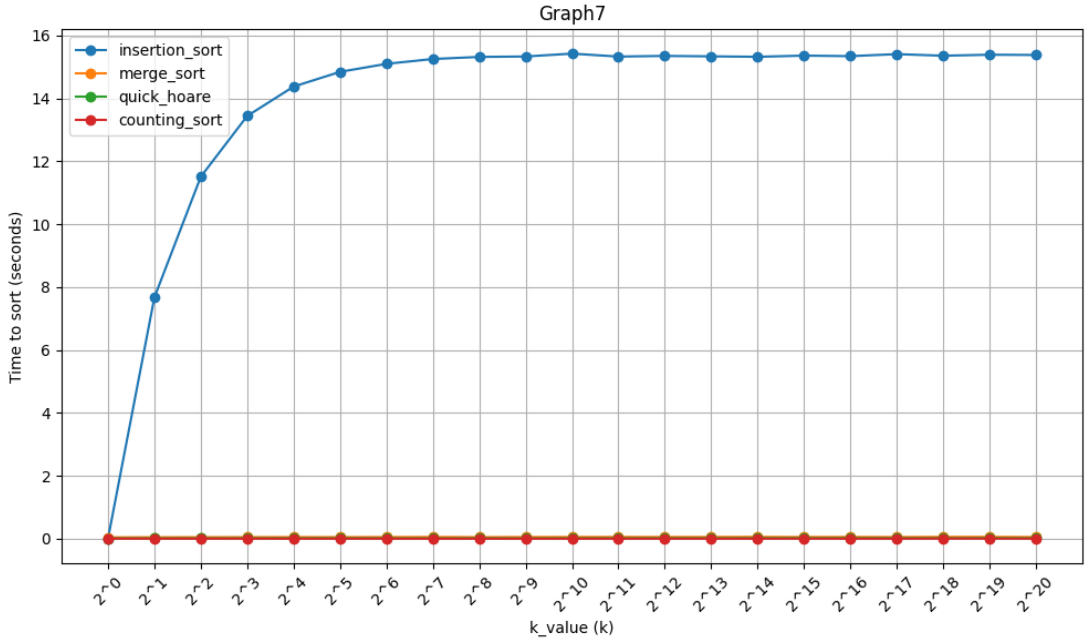
6. 第六張折線圖：



分析：

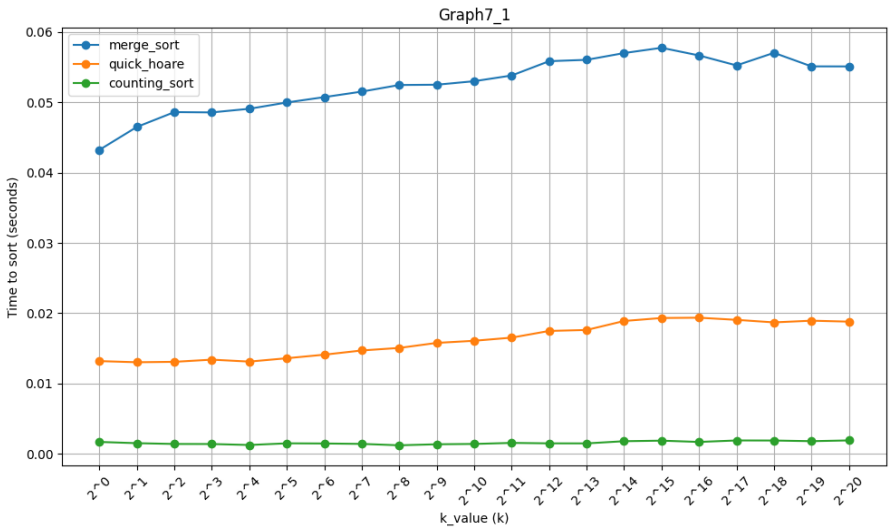
如圖五所提及，Quick sort(Hoare)與 Merge sort 並不會大幅受到陣列無序性影響時間，故個折線狀況相對平穩，且大小關係如同圖三提及的，當主要使用 Quick Sort 處理較大區段時，它可以更快地完成排序，也因此會有大部分 Quick_merge 較 Merge_quick 快。

7. 第七張折線圖：



觀察 csv 後發現，由於 insertion_sort 數值過大，壓縮了其他資料結構的顯示，所以我做了一個去除 insertion_sort 的版本。

折線圖(without insertion)：



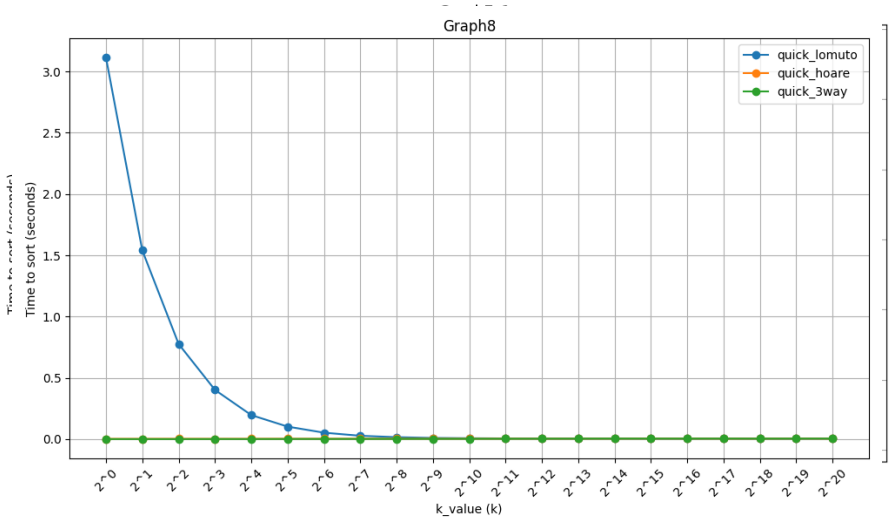
分析：

隨著 k 值增加，這使得陣列中的元素分佈更加分散和無序，且相對從已排序去做 k 次交換，這邊的 k 值增加會更快的提升陣列的無序性(隨機一次+2 與*2 的差別)。故當 k 值很小時所有元素都是相同的，陣列幾乎是排序的，Insertion Sort 可以很快完成排序(O(n))。但隨著 k 的增加，陣列的無序性也增加，

Insertion Sort 在排序時需要執行更多的比較和交換操作，導致執行時間急劇上升(O(n2))。

從圖 7_1 可發現，如同第四張圖分析，對於 Merge sort(無論 Input Array 皆為 O(nlogn))、Quick sort(取決於 pivot 選擇，但此次皆為 Randomized pivot)、Counting sort(無論 Input Array 皆為 O(n+k))並不會因為陣列是否接近排序好的陣列，而大幅影響到執行所需時間。此三種的所需時間大小關係則如同圖 1 中分析的為 counting sort 最快，最慢的是 merge sort。

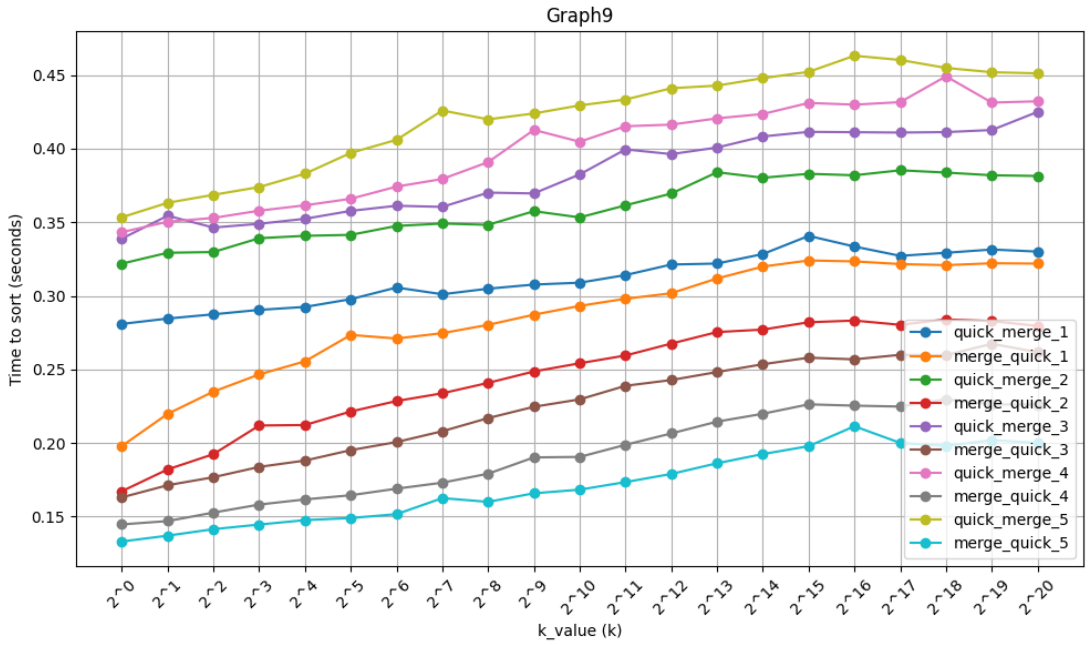
8. 第八張折線圖：



分析：

Lomuto 的執行時間隨著 k 值的增加而急劇下降。當 k 值小時，資料集中可能有許多重複的元素，這導致 Lomuto 會出現許多多餘的 swap 操作而增加許多時間。相對而言，Hoare 與 3way 方法就不用有這樣的問題，而導致不會在 k 值小時需有許多時間。

9. 第九張折線圖：



分析：

如同圖 6 與圖 8 中提及的，Quick sort(Hoare)與 Merge sort 並不會大幅受到陣列無序性影響時間而當主要使用 Quick Sort 處理較大區段時，它可以更快地完成排序，也因此也會有大部分 Quick_merge 較 Merge_quick 快。

問題：

在製作圖7時，我在原先設定陣列大小為2^{20}的時候，跑k=2時，我原先認為時間一樣會如同k=1一樣跑很快，但很快就很像進入無限迴圈出不來，後來就會直接跳出沒有輸出結果，也沒有跳出錯誤訊息，但在陣列調小後就沒有這個問題，能夠正常實驗。認知中Insertion sort雖然在陣列很大時會比較慢，但也會造訪玩arr[j]，並不會出現跑不出來的問題，故不知的這裡沒有輸出時間的原因。

```
PS C:\Users\USER\vscode workspace\NCCUCP\data_structure\2024DS_HW\hw2> python .\plot_script7.py
Running tests for k_value = 1
Running tests for k_value = 2
Test for k = 2 timed out!
Skipping remaining points for insertion_sort after k_value 2
```

(實驗時直接超出時間進行skip，但也沒成功輸出sort時間)