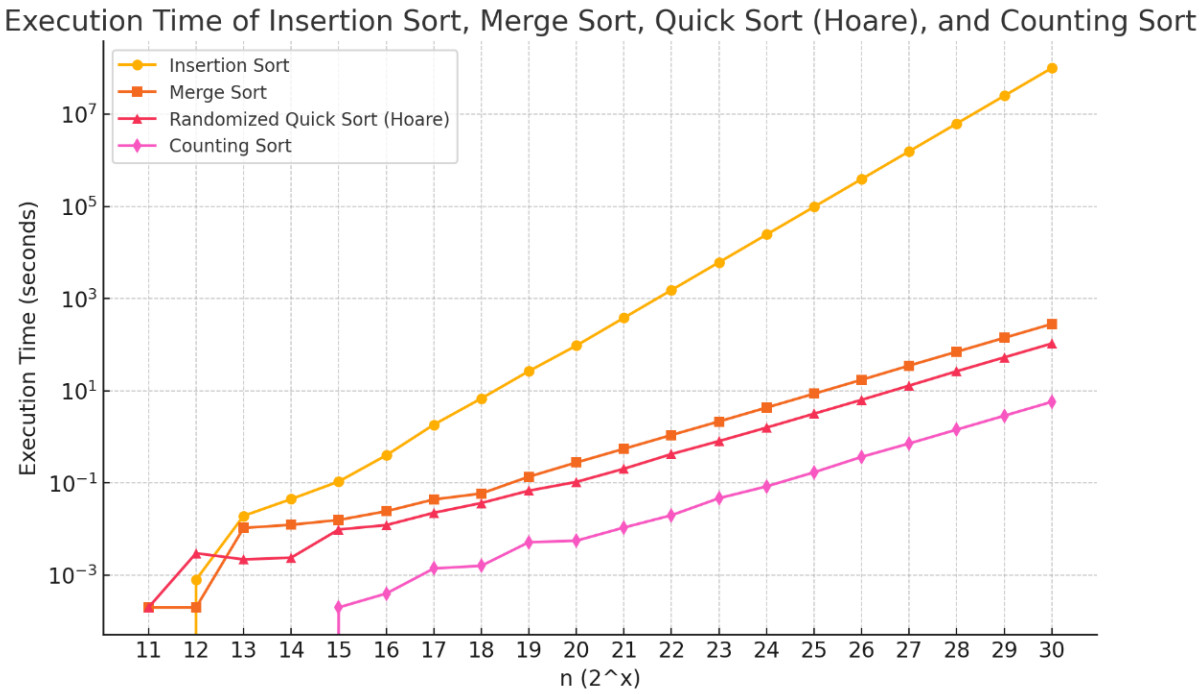
112703003 資訊二 黃柏淵 DS_HW2

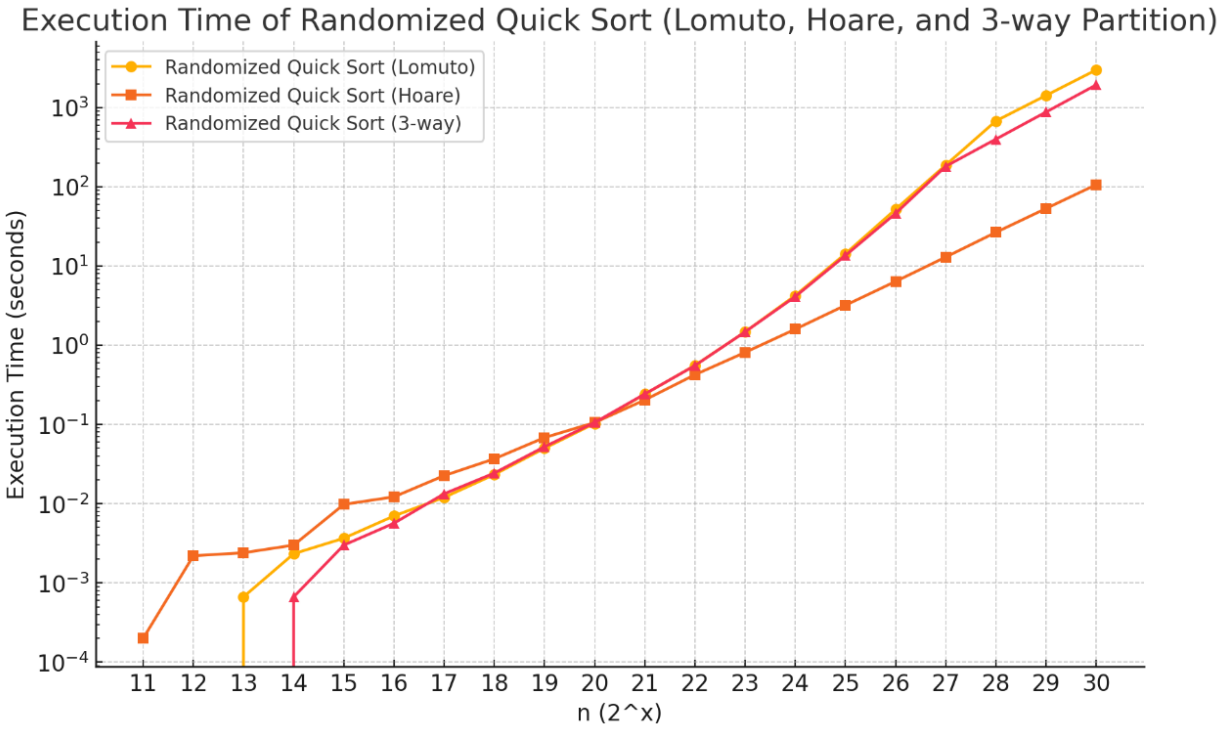
第一大題：實驗折線圖

圖一



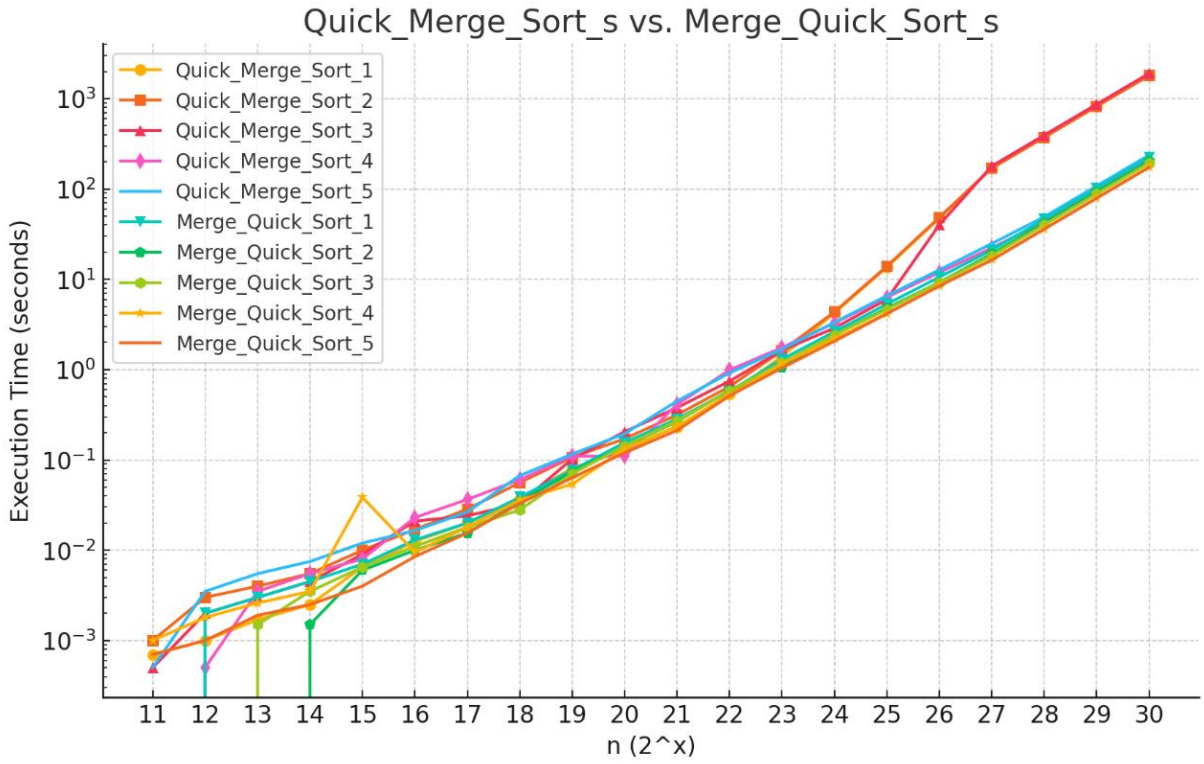
- Insertion Sort: 在 2 的 23 次方時，開始使用估計，因為 Insertion Sort 的時間複雜度是 O(n^2)。Insertion Sort 對於每新增的元素都需要與前面的元素比較，而進行大量的移動操作。執行到大規模的數據時，比較次數和移動次數都會急劇上升。如果我們將輸入大小 n 增加一倍，執行時間就會變為原來的四倍（因為 (2n)^2 = 4n^2）。因此推論的執行時間為 4 倍。
- Merge Sort: 在 2 的 28 次方使用估計。他的時間複雜度是 O(n log n)，執行時間大約通常會增加 1.5 到 2.2 倍，而我選擇用增加 2.1 倍去估計。
- Quick Sort (Hoare Partition): 在 2 的 29 次方時使用估計。平均時間複雜度為 O(n log n)。因此推論時間選用 2.1 倍。
- Counting Sort: 時間複雜度為 O(n + k)，在我的程式碼中，k ≈ n，因此時間複雜度可以簡化為 O(n)。隨著陣列大小 n 的增長，計數排序的執行時間將近似於線性增長。當 n 增加到兩倍時，執行時間理論上也將增加約 2 倍。

圖二



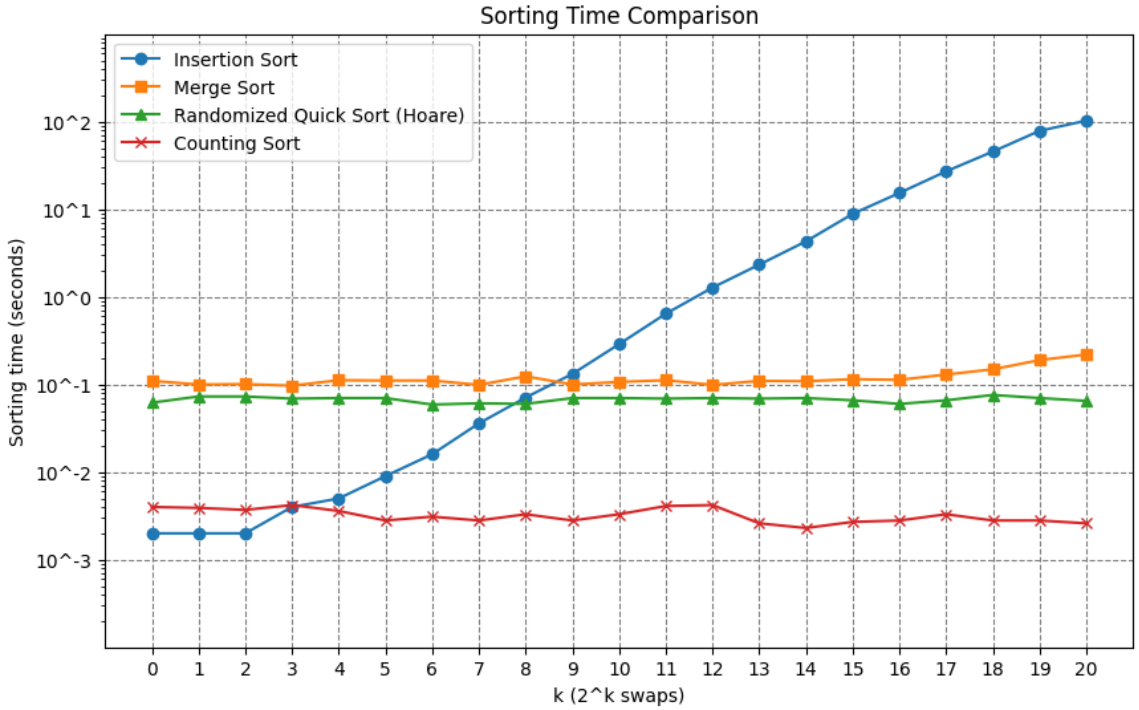
- Quick Sort (Lomuto Partition): 在 2 的 28 次方時使用估計，因為平均時間複雜度為 O(n log n)，執行時間大約通常會增加 1.5 到 2.2 倍，因此選擇用 2.1 倍去估計。
- Quick Sort (Hoare Partition): 在 2 的 29 次方時使用估計。平均時間複雜度為 O(n log n)。因此推論時間選用 2.1 倍。
- Quick Sort (3 way partition): 在 2 的 28 次方時使用估計。平均時間複雜度為 O(n log n)。因此推論時間選用 2.1 倍。

圖三



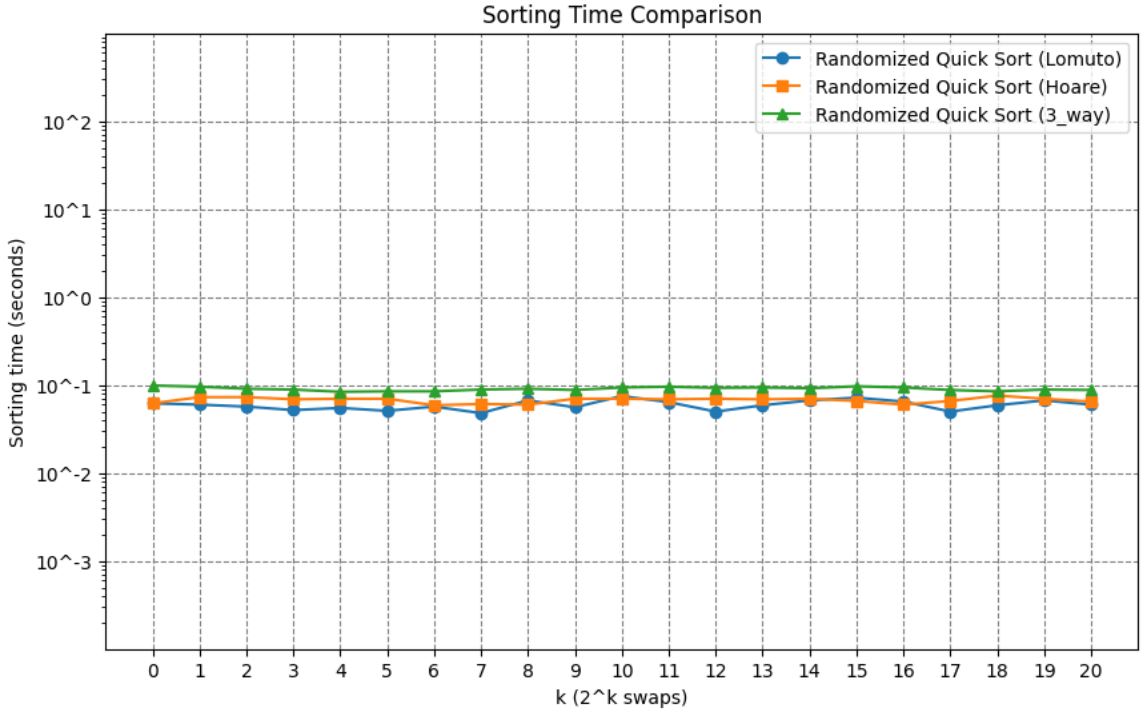
- Quick_Merge_Sort_s: 在 2 的 28 次方時使用到估計，Quick Sort 的平均時間複雜度為 O(n log n)，Merge Sort 的時間複雜度為 O(n log n)。當 threshold 設置得較大時，大部分數據將使用 Merge Sort，總體時間複雜度會傾向於 O(n log n)。如果 threshold 較小，將以 Quick Sort 為主，平均情況下也會接近 O(n log n)，因此推論的時間採用 2.1 倍。
- Merge_Quick_Sort_s: 在 2 的 28 次方使用到估計，Merge Sort 的時間複雜度為 O(n log n)， Quick Sort 的平均時間複雜度為 O(n log n)。當 threshold 設置得較大時，大部分數據將使用 Quick Sort，總體時間複雜度會傾向於 O(n log n)。如果 threshold 較小，將以 Merge Sort 為主，平均情況下也會接近 O(n log n)，因此推論的時間採用 2.1 倍。

圖四



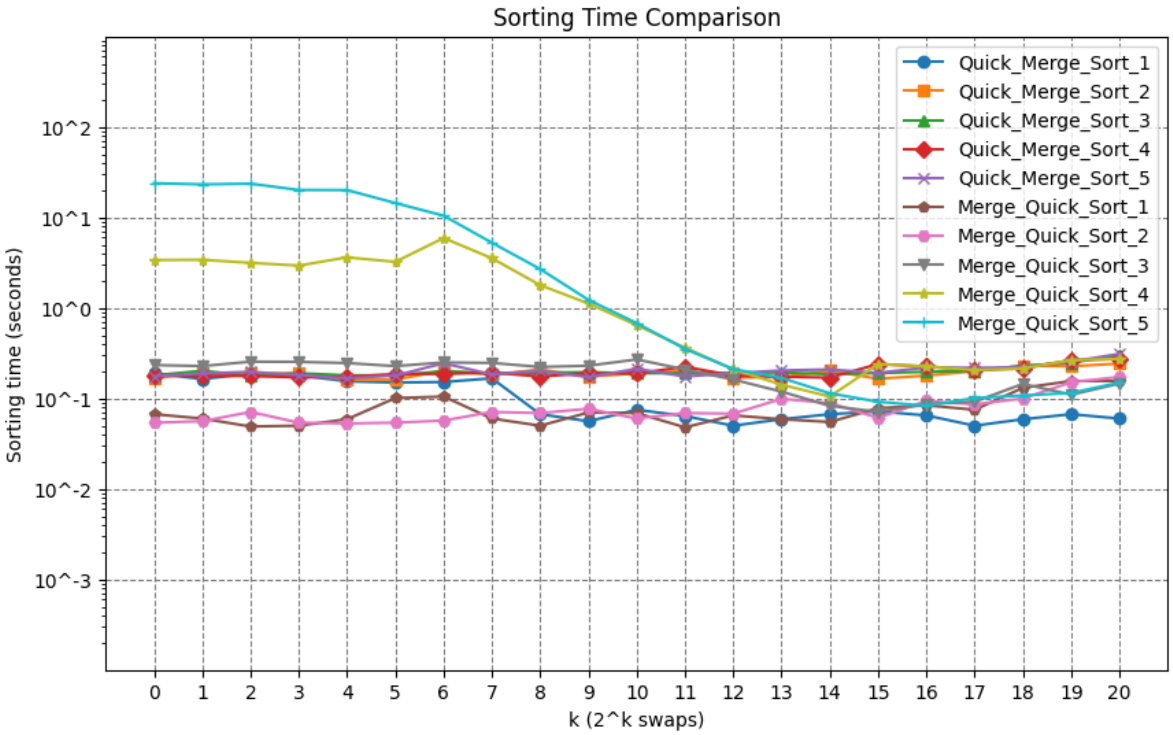
- 未使用估計

圖五



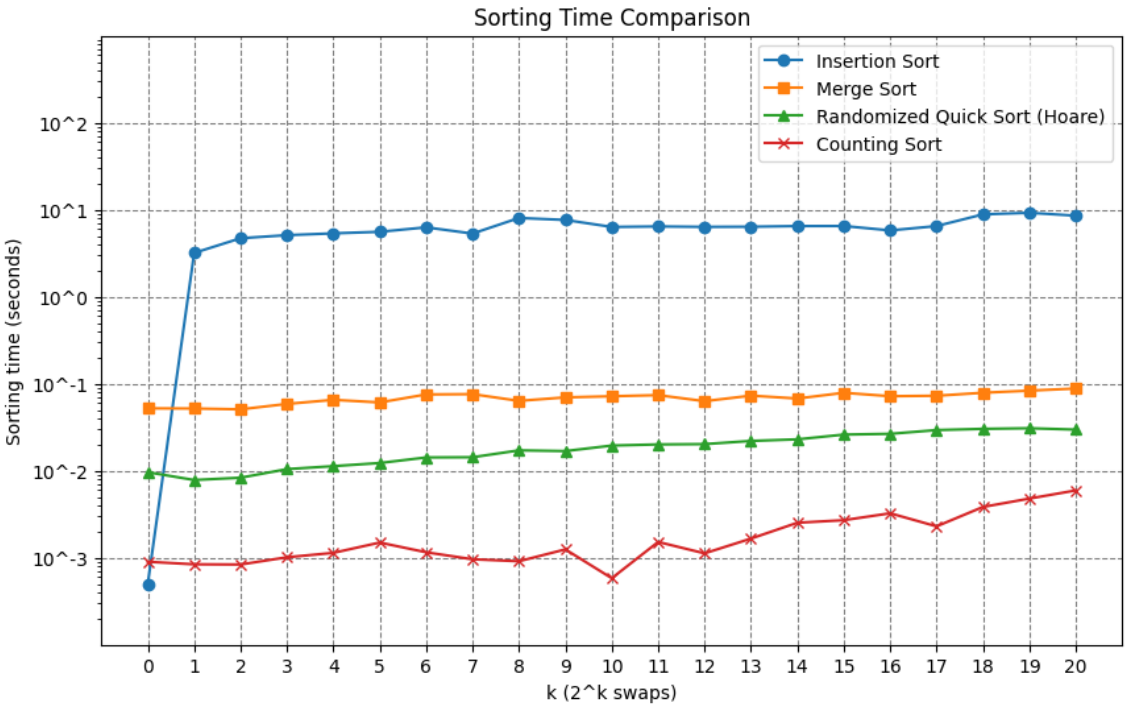
- 未使用估計

圖六



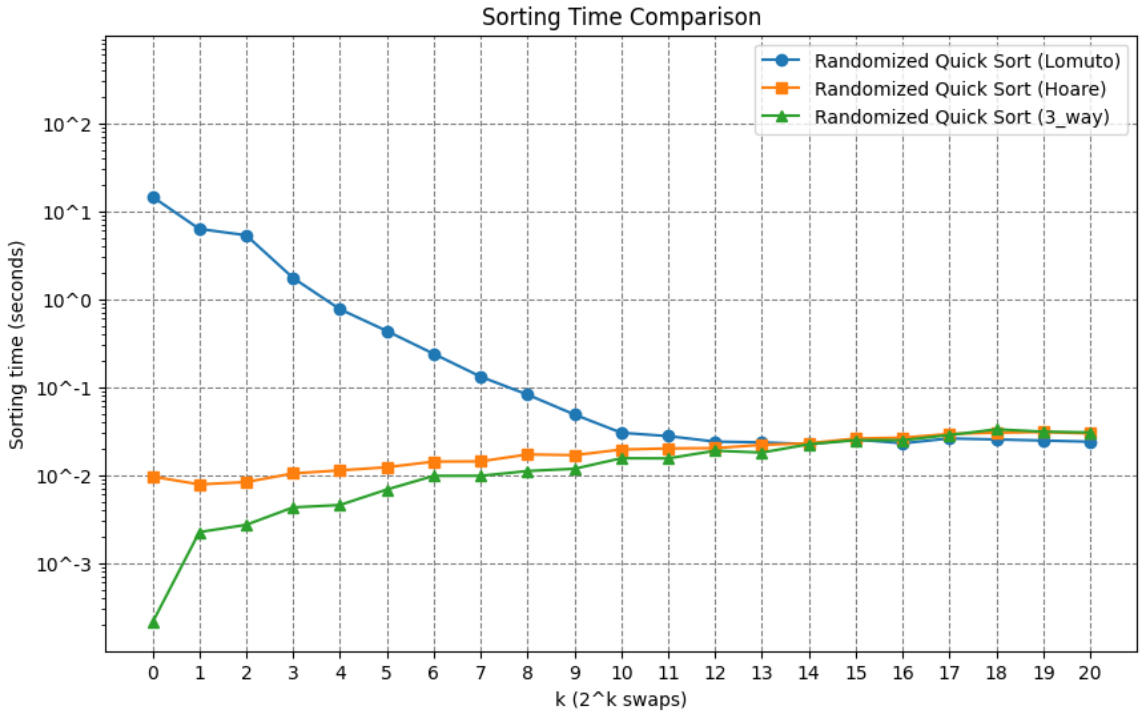
- 未使用估計

圖七



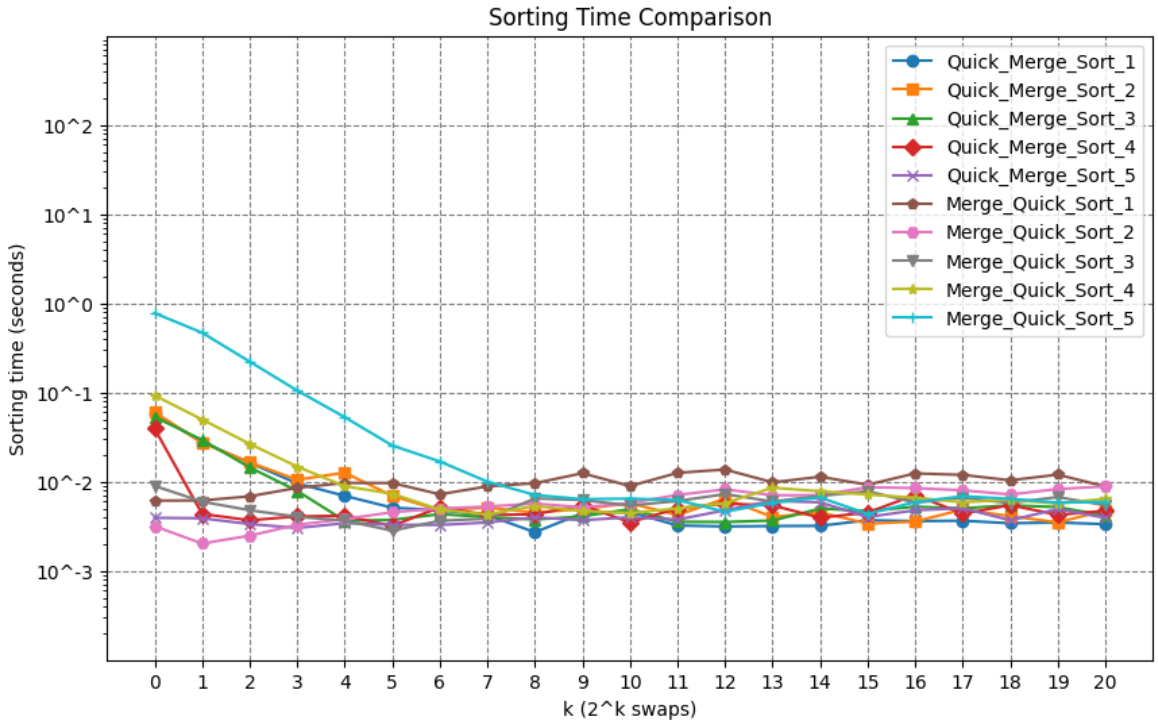
- 未使用估計。array 大小改成 2^18 次方

圖八



- 未使用估計。array 大小改成 2^18 次方

圖九



- 未使用估計。array 大小改成 2^18 次方

第二大題：排序演算法程式碼與解釋

1. Insertion Sort

```cpp
// 插入排序算法
void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; 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;
    }
}
```

i. 遍歷陣列，每次將當前元素 key 插入到前面已排序的部分，使已排序部分保持有序。

ii. 插入元素：對於每個 key，從已排序部分的最後開始，將比 key 大的元素向右移動，為 key 留出合適的位置。當找到 key 的插入位置時，將 key 插入。

2. Merge Sort

```cpp
// 合併排序的合併步驟
void merge(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (int i = 0; i < temp.size(); ++i) arr[left + i] = temp[i];
}

// 合併排序的遞迴部分
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

- void merge()
  i. 將 arr 的 [left, mid] 和 [mid+1, right] 兩個已排序的子陣列合併成一個排序的整體。
  ii. 使用 temp 陣列臨時儲存合併結果，再將結果複製回 arr 的對應位置。

- void mergeSort()
  i. 不斷將陣列分割成左右兩部分，直到每個子陣列只有一個元素。
  ii. 遞迴地調用 merge 函式將已排序的子陣列合併。

3. Counting Sort

```
// 計數排序
void countingSort(vector<int>& arr, int maxValue) {
    vector<int> count(maxValue + 1, 0);

    // 計算每個元素出現的次數
    for (int num : arr) {
        count[num]++;
    }

    // 根據 count 陣列重建排序後的 arr
    int index = 0;
    for (int i = 0; i <= maxValue; ++i) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
```

- vector<int>& arr：待排序的整數向量。
- int maxValue：數組中最大元素的值，用來確定計數陣列的大小。
- vector<int> count(maxValue + 1, 0)：建立計數陣列 count，大小為 maxValue + 1，初始值為 0。這個陣列用來記錄每個數字在 arr 中出現的次數。
- for (int num : arr)：遍歷 arr 中的每一個元素。
- count[num]++：將元素 num 在 count 陣列中對應的位置加 1，以記錄 num 出現的次數。
- int index = 0：用於追蹤 arr 中當前的位置。
- for (int i = 0; i <= maxValue; ++i)：遍歷 count 陣列的每一個位置，對應數值從 0 到 maxValue。
- while (count[i] > 0)：只要 count[i] 大於 0，就將 i 放入 arr 中。
- arr[index++] = i：將數值 i 插入 arr 的當前索引位置，並將 index 增加 1。
- count[i]--：將 count[i] 減 1，表示已經處理過一個 i。

4. Quick Sort (Hoare Partition)

```cpp
// Hoare 分區算法
int hoarePartition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int left = low - 1;
    int right = high + 1;

    while (true) {
        do {
            left++;
        } while (arr[left] < pivot);

        do {
            right--;
        } while (arr[right] > pivot);

        if (left >= right) return right;

        std::swap(arr[left], arr[right]);
    }
}

// 隨機選擇基準點的 Quick Sort
void randomizedQuickSort(std::vector<int>& arr, int low, int high, std::mt19937& gen) {
    if (low < high) {
        std::uniform_int_distribution<> dist(low, high);
        int randomPivotIndex = dist(gen);
        std::swap(arr[low], arr[randomPivotIndex]); // 將隨機選定的基準移到開頭

        int pivotIndex = hoarePartition(arr, low, high);
        randomizedQuickSort(arr, low, pivotIndex, gen);
        randomizedQuickSort(arr, pivotIndex + 1, high, gen);
    }
}
```

Hoare Partition，用於根據 pivot 將陣列分為兩部分：小於 pivot 的部分和大於 pivot 的部分。它返回分割後的 index，用於快速排序的後續遞迴。

- int hoarePartition()
  i. 選擇基準點：將 arr[low] 設為 pivot。
  ii. 初始化指標：left 從 low - 1 開始，right 從 high + 1 開始。
  iii. 尋找交換元素：
    a. left 向右移動，直到找到一個不小於 pivot 的元素。
    b. right 向左移動，直到找到一個不大於 pivot 的元素。
  iv. 交換與結束：
    a. 如果 left 和 right 相遇或交錯，返回 right 作為分割的 index。
    b. 否則，交換 arr[left] 和 arr[right]，繼續分區。

- void RandomizedQuickSort()
  i. 遞迴結束條件：當 low 不小於 high 時，返回。
  ii. 隨機選擇 pivot：

a. 使用 std::uniform_int_distribution 和 std::mt19937 隨機生成 low 到 high 範圍內的一個 index。
b. 將隨機基準點與 arr[low] 交換，使其成為分區的 pivot。

iii. 分區：
a. 調用 hoarePartition 函式，分割陣列並獲得分割的 index - pivotIndex。

iv. 遞迴排序：
a. 對基準點左側的子陣列（low 到 pivotIndex）和右側的子陣列（pivotIndex + 1 到 high）分別進行遞迴排序。

5. Quick Sort (Lomuto Partition)

```cpp
// Lomuto 分區的 Quick Sort
int lomutoPartition(vector<int>& arr, int left, int right) {
    int pivot = arr[right];
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[right]);
    return i + 1;
}

void randomizedQuickSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        // 隨機選取一個 pivot 並將其移動到結尾
        int randomPivotIndex = left + rand() % (right - left + 1);
        swap(arr[right], arr[randomPivotIndex]);

        // 使用 Lomuto 分區
        int p = lomutoPartition(arr, left, right);

        // 遞迴進行 Quick Sort
        randomizedQuickSort(arr, left, p - 1);
        randomizedQuickSort(arr, p + 1, right);
    }
}
```

- lomutoPartition ()：
  i. 選擇 pivot：以陣列最右邊的元素 arr[right] 作為基準點。
  ii. 初始化指標：變數 i 初始設為 left - 1，表示較小數字的範圍邊界。
  iii. 遍歷並分區：
    a. 遍歷從 left 到 right - 1 的元素，如果 arr[j] 小於基準點，則將其交換至 i 的右邊，並增加 i。
  iv. 移動 pivot：遍歷結束後，將 pivot 放在 i + 1 位置，將小於基準點的數字放在基準點左邊，大於基準點的數字放在右邊。
  v. 返回分區點：最後，i + 1 作為分區點返回，以便在 Quick Sort 中進

行左右子陣列排序。

- randomizedQuickSort ()：
  i. 當 left 小於 right 時繼續遞迴。這是 Quick Sort 的遞迴結束條件。
  ii. 隨機選取基準點：
    a. 透過 rand() 函數隨機選取一個 pivot Index。
    b. 將選定的基準點與最右邊的元素交換，這樣便能在 lomutoPartition 中使用隨機基準點。
  iii. 呼叫 lomutoPartition，將陣列分成兩部分並取得分區點 p。
  iv. 遞迴 Quick Sort：對基準點左側 (left 到 p-1) 和右側 (p+1 到 right) 分別遞迴進行 Quick Sort，完成排序。

6. Quick Sort (3 way Partition)

```cpp
void threeWayPartition(vector<int>& arr, int low, int high, int& lt, int& gt) {
    int pivot = arr[low]; // 使用最左邊的元素作為樞軸
    lt = low;             // lt 為小於樞軸部分的邊界
    gt = high;            // gt 為大於樞軸部分的邊界
    int i = low + 1;      // 掃描當前元素的指標

    while (i <= gt) {
        if (arr[i] < pivot) {
            swap(arr[lt], arr[i]);
            lt++;
            i++;
        } else if (arr[i] > pivot) {
            swap(arr[i], arr[gt]);
            gt--;
        } else {
            i++;
        }
    }
}

// 隨機化三向分區
void randomizedThreeWayPartition(vector<int>& arr, int low, int high, int& lt, int& gt) {
    int randomPivotIndex = low + rand() % (high - low + 1);
    swap(arr[low], arr[randomPivotIndex]); // 隨機選擇樞軸並將其換到最左邊
    threeWayPartition(arr, low, high, lt, gt); // 使用三向分區法
}

// 隨機化三向快速排序
void randomizedQuickSort3Way(vector<int>& arr, int low, int high) {
    if (low < high) {
        int lt, gt;
        randomizedThreeWayPartition(arr, low, high, lt, gt);
        randomizedQuickSort3Way(arr, low, lt - 1);  // 對小於樞軸部分排序
        randomizedQuickSort3Way(arr, gt + 1, high); // 對大於樞軸部分排序
    }
}
```

threeWayPartition ()：

- 選擇 pivot：選取陣列的最左邊元素作為基準點。
- 初始化邊界：
  i. lt 表示小於基準點的元素範圍邊界，初始化為 low。
  ii. gt 表示大於基準點的元素範圍邊界，初始化為 high。
  iii. i 是當前掃描元素的指標，初始值為 low + 1。
- partition：
  i. 當 i <= gt 時進行掃描和交換操作：
    a. 如果 arr[i] < pivot，表示該元素屬於小於基準點的部分，將其與 arr[lt] 交換，並將 lt 和 i 向右移動。
    b. 如果 arr[i] > pivot，表示該元素屬於大於基準點的部分，將其與 arr[gt] 交換，並將 gt 向左移動。
    c. 如果 arr[i] == pivot，表示該元素等於基準點，只需將 i 向右移動。
  ii. 結束時，所有小於基準點的元素位於 [low, lt-1]，等於基準點的元素位於 [lt, gt]，大於基準點的元素位於 [gt+1, high]。

randomizedThreeWayPartition ()：

- 隨機選取 pivot：
  i. 隨機選擇 index 作為基準點，從 low 到 high 範圍內。
  ii. 將選定的基準點與最左邊元素交換，以便在 threeWayPartition 中使用這個隨機基準點。
- Three way partition：
  i. 呼叫 threeWayPartition 函數，進行分區，確定小於、等於和大於基準點的元素範圍。

randomizedQuickSort3Way ()：

- 當 low < high 時才進行遞迴，這是 Quick Sort 的遞迴結束條件。
- Three way partition：
  i. 呼叫 randomizedThreeWayPartition 函數，將陣列分為小於基準點、等於基準點、大於基準點三部分，並得到 lt 和 gt 的分界範圍。
- 遞迴 Quick Sort：
  i. 對 [low, lt - 1] 部分進行遞迴排序。
  ii. 對 [gt + 1, high] 部分進行遞迴排序。

7. Quick_Merge_Sort_s (Quick Sort 採用 Hoare Partition)

```
// Hybrid Quick_Merge_Sort_s algorithm with threshold for switching to Merge Sort
void quickMergeSort(vector<int>& arr, int left, int right, int threshold) {
    if (left < right) {
        // If the segment size is small enough, use Merge Sort
        if ((right - left + 1) <= threshold) {
            mergeSort(arr, left, right);
        } else {
            // Randomly select a pivot and use Hoare's partition scheme
            int randomPivotIndex = left + rand() % (right - left + 1);
            swap(arr[left], arr[randomPivotIndex]);
            int p = hoarePartition(arr, left, right);

            // Recursive Quick Sort calls
            quickMergeSort(arr, left, p, threshold);
            quickMergeSort(arr, p + 1, right, threshold);
        }
    }
}
```

參數：

i. arr：要排序的數列。

ii. left 和 right：分區的左右邊界。

iii. threshold：設定的閾值，用來判斷何時應切換到 Merge Sort。

iv. 遞迴終止條件：當 left >= right 時不再遞迴，表示區段內僅剩一個元素或無元素，已自然排序。

判斷是否使用 Merge Sort：

i. 當分區大小 (right - left + 1) 小於或等於 threshold 時，算法切換到 Merge Sort。

若區段大小大於 threshold，則使用 Quick Sort：

i. 隨機選取基準點：選擇 left 到 right 範圍內的隨機索引作為基準點 randomPivotIndex，將該基準點與 left 元素交換。

ii. 使用 Hoare 分區法：呼叫 hoarePartition 函數，以基準點將數列劃分為左右兩部分。

iii. 呼叫 quickMergeSort 對左側和右側區段分別進行遞迴排序：

a. quickMergeSort(arr, left, p, threshold); 對左側區段進行排序。

b. quickMergeSort(arr, p + 1, right, threshold); 對右側區段進行排序。

8. Merge_Quick_Sort_s (Quick Sort 採用 Hoare Partition)

```
// Modified Merge_Quick_Sort algorithm with threshold for switching to Quick Sort
void modifiedMergeQuickSort(vector<int>& arr, int left, int right, int threshold) {
    if (left < right) {
        // If the segment size is small enough, use Quick Sort
        if ((right - left + 1) <= threshold) {
            quickSort(arr, left, right);
        } else {
            int mid = left + (right - left) / 2;

            // Recursive calls to divide the array
            modifiedMergeQuickSort(arr, left, mid, threshold);
            modifiedMergeQuickSort(arr, mid + 1, right, threshold);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }
}
```

參數：

i. arr：要排序的數列。left 和 right：分區的左右邊界。threshold：閾值，用來判斷何時切換到 Quick Sort。

條件：

i. 當 left >= right 時不再進行遞迴，達到終止條件，表示區段中只剩下一個或沒有元素。

判斷是否使用 Quick Sort：

i. 當分區大小 (right - left + 1) 小於或等於 threshold 時，切換到 Quick Sort。呼叫 quickSort(arr, left, right); 將指定區段排序。

遞迴分割陣列：

i. 若大於 threshold，使用 Merge Sort 繼續進行排序。

ii. 遞迴劃分區段：

a. 使用 mid = left + (right - left) / 2 計算中間 index，將 arr[left..right] 分割成左右兩部分。

b. 遞迴處理左半部分：呼叫 modifiedMergeQuickSort(arr, left, mid, threshold)。

c. 遞迴處理右半部分：呼叫 modifiedMergeQuickSort(arr, mid + 1, right, threshold)。

合併排序區段：

i. 當左右兩部分已經經過 Quick Sort 或 Merge Sort 排序後，呼叫 merge(arr, left, mid, right); 合併已排序的兩部分。

第三大題：解釋實驗圖

圖一：Insertion Sort, Merge Sort, Q_Sort(Hoare Partition)及 Counting Sort 在陣列大小範圍從 2^10 到 2^30，每次排序的時間。

1. Insertion Sort: 因為平均時間複雜度是 O(n^2)，因此當我給定的數值 2 倍增加，排序的時間會變成 4 倍。
2. Merge Sort: 因為題目有對時間複雜度有要求 O(n log n)，因此我的時間推估是用 2.1 倍。
3. Q_Sort(Hoare): 因為平均時間複雜度是 O(n log n)，因此當我給定的數值 2 倍增加，排序的時間會變成約 2.1 倍。跟 Merge Sort 相同。
4. Counting Sort: 其時間複雜度是 O(n + k)，因為我的程式碼會使得 k 約等於 n，因此可以讓時間複雜度變成 O(n)，當我給定的數值 2 倍增加，排序的時間會變成 2 倍。而它的成長幅度也是四條折線中最慢的，在圖中是最下面的那條。

因此在這張圖中，Insertion Sort 會成長幅度最多，Merge Sort 跟 Q_Sort(Hoare)成長幅度相同，而 Counting Sort 的所花的時間最少，也呈現線性增長。

圖二：Q_Sort(Lomuto, Hoare, 3_Way Partition) 在陣列大小範圍從 2^10 到 2^30，每次排序的時間。

1. Lomuto: 它的時間複雜度平均情況是 O(n log n)，因此當我給定的數值 2 倍增加，排序的時間會變成約 2.1 倍。對於大陣列，由於 Lomuto 只將元素分為「小於基準點」和「大於等於基準點」兩部分，如果基準點選擇不佳，則會導致不均衡分區，使得時間複雜度退化，因此當 array 變大，它的時間是最多的。
2. Hoare: 因為平均時間複雜度是 O(n log n)，因此當我給定的數值 2 倍增加，排序的時間會變成約 2.1 倍。Hoare 分區通常能夠產生更平衡的分區，與 Lomuto 分區相比，Hoare 分區更適合處理大陣列，因此當陣列變大，時間較少。
3. 3_Way: 它的時間複雜度平均情況是 O(n log n)，因此當我給定的數值 2 倍增加，排序的時間會變成約 2.1 倍。

因此在這張圖中，三條折線的成長幅度略同，當 arr 大小變大時，Lomuto 的時間會花最多，而 Hoare 的時間會花最少，但在 arr 大小較小的時候，Hoare 花的時間卻比較多。

圖三：Quick_Merge_Sort_s (s=1,2,3,4,5)跟 Merge_Quick_Sort(s=1,2,3,4,5) 在陣列大小範圍從 2^10 到 2^30，每次排序的時間。

在這張圖中，十條線的成長幅度都略同，只有當 arr 大小變大時，Quick_Merge_Sort_(1,2,3) 所花的時間會增長較多，因為當 s 是 1, 2, 3 時，threshold 是較小的數值，使 Quick_Merge_Sort 在進行遞迴分割的過程中更早地切換到 Merge Sort。而 Merge Sort 的時間複雜度是 O(n log n) ，但其操作次數通常比 Quick Sort 多。因此，當切換點更早且 Merge Sort 次數增加時，操作時間會較高。當 s 是 4, 5 時，threshold 非常大，所以 Quick Sort 在分割陣列時不會輕易切換到 Merge Sort，而是繼續執行 Quick Sort。Quick Sort 在一般情況下的平均時間複雜度也是 O(n log n)，但其常數較低，效能較好。當使用 Quick Sort 分割大部分陣列後，僅剩非常小的子陣列才會使用 Merge Sort，因此操作時間較短。

圖四：Insertion sort、merge sort、randomized quick sort with Hoare partition、counting sort 在陣列大小為 2^20 下，隨機 swap array 內容 k 次，重新排序所需的時間。

1. Insertion Sort: 當 k 的值越大時，它的時間複雜度會接近 O(n^2)，因此成長幅度提升。
2. Merge Sort: 它時間複雜度是 O(n log n)，雖然每次迴圈都增加了隨機交換次數 k，但這對合併排序的性能影響極小。因為按照「分割 - 合併」的流程來執行，操作數量僅由資料大小 n 決定。因此即使進行了大量隨機交換，Merge Sort 的執行時間仍然保持穩定。
3. Q_Sort(Hoare): 它時間複雜度是 O(n log n)，Hoare Partition 在每次分割時不需要將 pivot 放置在最終位置，因此它的分割過程比較高效，使得執行時間會保持穩定。
4. Counting Sort: 它的時間複雜度是 O(n + k)，當 k 略等於 n 時，它的複雜度會變成 O(n)，因此它的執行時間是四條折線中最少的，因為的執行時間只依賴於陣列大小 n 和最大值範圍 max value，當我把 arr 大小控制在 2^20 時，執行時間也會保持穩定。

因此在這張圖中，只有 Insertion Sort 的執行時間會呈現成長，其他三條折線保持穩定(水平)，而那三條線中，Counting Sort 所花的時間是最少的。

圖五：Q_Sort(Lomuto, Hoare, 3_Way Partition) 在陣列大小為 2^20 下，隨機 swap array 內容 k 次，重新排序所需的時間。

1. Lomuto: 隨機交換次數 k 的增加會使陣列更無序，但這對 Quick Sort 的平均情況影響有限。隨著無序度增加，Quick Sort 的 partition 步驟和效率並不會顯著落差，因此排序時間仍然能穩定在 O(n log n)。
2. Hoare: 它時間複雜度是 O(n log n)，Hoare Partition 在每次分割時不需要將 pivot 放置在最終位置，因此它的分割過程比較高效，使得執行時間會保持穩定。
3. 3_Way: 隨機選擇 pivot 可以有效減少 Quick Sort 的最壞情況 O(n^2)，保持時間複雜度穩定在 O(n log n)，也因為它把等於 pivot 的元素分開處理，減少了遞迴深度和排序操作的次數，使得時間保持穩定。

因此在這張圖中，三條折線的時間皆呈現穩定狀態(水平)。

圖六：Quick_Merge_Sort_s (s=1,2,3,4,5)跟 Merge_Quick_Sort(s=1,2,3,4,5) 在陣列大小為 2^20 下，隨機 swap array 內容 k 次，重新排序所需的時間。

這張圖可以發現 Merge_Quick_Sort_(4, 5)的一開始的執行時間是遠高於其他，而過了 2^4 後，開始遞減，並在 2^12 左右回歸到穩定狀態。當 s = 4, 5 時，threshold 值較大，程式更早切換至 Quick Sort，由於 Quick Sort 對於無序度較高的數據執行時間較長，因此在 k 值較小時（陣列更接近有序），執行時間較高。隨著 k 增加，Quick Sort 的切分次數減少，加上換回 Merge Sort 進行排序，因此執行時間會先在高處，然後隨著 k 的增加而逐漸減少。而另外八條折線的執行時間都趨於穩定。

圖七：Insertion sort、merge sort、randomized quick sort with Hoare partition、counting sort 在 array 大小為 2^20(arr[i] = 0~k-1 隨機產生)下，測量排序時間。

1. Insertion Sort: 它的時間複雜度是 O(n^2)，在 k 越來越大的時候，執行時間有小幅度的提高。
2. Merge Sort: 它的時間複雜度是 O(n log n)，這個複雜度在多數情況下不受資料初始排列順序或無序程度的影響，因此執行時間呈現接近穩定狀態。
3. Q_Sort(Hoare): 它的時間複雜度是 O(n log n)，透過隨機選擇 pivot，可以有效避免最壞情況，因此降低了因陣列初始無序度帶來的負面影響，增加 k 的值只是改變了隨機生成的數字範圍，這並不顯著改變 Quick Sort 的分區步驟數量和操作數量，因此排序時

間維持穩定。

4. Counting Sort: 它的時間複雜度是 O(n + k)，k 只會影響記錄每個數字出現次數的輔助陣列(簡稱：計數陣列)的大小。對於大多數情況，計數陣列的構建時間比排序步驟的時間影響要小，因此執行時間會主要取決於 O(n)的部分。由於計數排序不涉及比較，效率較高。而執行時間呈現接近穩定狀態。

這張圖可以發現 Insertion Sort 的執行時間會在 k = 0 到 k = 1 的時候增加很多，然後就趨於穩定在四條折線的最上方(最花時間)，再來第二花時間的則是 Merge Sort，再來是 Quick Sort(Hoare)，花最少時間的是 Counting Sort。

圖八：Q_Sort(Lomuto, Hoare, 3_Way Partition) 在 array 大小為 2^20(arr[i] = 0~k-1 隨機產生)下，測量排序時間。

1. Lomuto: 一開始的時間是所有折線裡面最高的，然後當 k 增加時，執行時間開始減少。因為當 k 值較小時，數據中可能會出現大量重複元素。隨機生成的數據範圍是 0 到 k - 1，所以範圍較小的 k 將產生較多的重複數據，增加排序過程中的比較次數。重複數據會導致 Lomuto 效率較低，執行時間也較久。隨著 k 增加，隨機數生成的範圍也增加，數據變得更均勻分布。均勻分布的數據通常更容易進行分區，因為分區點會更接近中間值，導致分割較為平衡的子數組，有助於減少遞歸深度並提高排序效率。因而時間會減少
2. Hoare: 它的時間複雜度是 O(n log n)，透過隨機選擇 pivot，可以有效避免最壞情況，因此降低了因陣列初始無序度帶來的負面影響，增加 k 的值只是改變了隨機生成的數字範圍，這並不顯著改變 Quick Sort 的分區步驟數量和操作數量，因此排序時間維持穩定。
3. 3_Way: 當 k 值較小時，隨機生成的數據範圍也很小，因此數組中會出現大量重複元素。3-way 對重複元素處理效率較高，因為它會將等於基準值的元素歸為一類，使排序操作減少了重複元素的排序開銷。當 k 增大時，隨機生成的數值範圍更大，使得數據分布更加均勻，重複元素的比例減少。3-way 的優勢減少，導致排序效率降低，並使得排序時間增加。

這張折線圖呈現的是 Lomuto 的時間是遞減狀態，而後保持穩定，Hoare 的是一直保持穩定，3_way 的是先增加，再保持穩定。

圖九：Quick_Merge_Sort_s (s=1,2,3,4,5)跟 Merge_Quick_Sort(s=1,2,3,4,5) 在 array 大小為 2^20(arr[i] = 0~k-1 隨機產生)下，測量排序時間。

Merge_Quick_Sort_s(4, 5)時，執行時間一開始會先減少，之後才趨於穩定。因為當 k 較小時（初期）數值範圍小，大量重複元素快速排序在處理重複元素時效率較低，Lomuto 會進行較多不必要的交換，大量相等元素會導致分區不平衡，這導致初期執行時間較長。k 增加時（中期），數值分布更均勻，重複元素減少，快速排序效率提升，分區更平衡，交換操作更有效，執行時間會下降。k 足夠大時（後期），數值幾乎不重複，快速排序達到最佳效能，執行時間趨於穩定。而其他條折線雖然有些微誤差，但因為數據無序度飽和後，算法的時間複雜度不再顯著增加，執行時間趨於穩定。

第四大題：遇到的問題

1. 當假設時間是遞減的狀態，如果陣列太大導致剛開始操作的時間會太長，我該選擇去調整陣列大小，還是從讓 k 值從 2^20 開始執行，並推論出遞增時間進行估值?
2. 終端機會出現 std::bad_alloc，並把程式終止，我把陣列大小做調整，但還是會出現。
3. 如果我一次一次取得實驗資料跟讓他跑 10 次並平均下來的資料會不會有差異?些微差異也算的話。
4. 圖七的 Insertion Sort 在 k=0 跟 k=1 的執行時間差異不合理(超大幅度增加)。