HW1

111511245 朱昱安

一、假設

在課堂中，我們學習到 Insertion Sort 的時間複雜度是$O(n) = n^2$，而 Merge Sort 的時間複雜度則是$O(n) = nlogn$。接下來利用 C++程式模擬兩種分類的結果：透過隨機產生一個不同 size 的 input array，並比較其執行時間。

二、程式結果

```
Testing with array size: 100
Insertion Sort time: 0 seconds
Merge Sort time: 0 seconds

Testing with array size: 500
Insertion Sort time: 0 seconds
Merge Sort time: 0 seconds

Testing with array size: 1000
Insertion Sort time: 0.001998 seconds
Merge Sort time: 0 seconds

Testing with array size: 2500
Insertion Sort time: 0.008201 seconds
Merge Sort time: 0.001256 seconds

Testing with array size: 5000
Insertion Sort time: 0.0355 seconds
Merge Sort time: 0.004063 seconds

Testing with array size: 7500
Insertion Sort time: 0.077273 seconds
Merge Sort time: 0.003284 seconds
```

```
Testing with array size: 10000
Insertion Sort time: 0.121288 seconds
Merge Sort time: 0.004673 seconds

Testing with array size: 25000
Insertion Sort time: 0.743966 seconds
Merge Sort time: 0.009519 seconds

Testing with array size: 50000
Insertion Sort time: 2.96421 seconds
Merge Sort time: 0.035724 seconds

Testing with array size: 75000
Insertion Sort time: 6.70466 seconds
Merge Sort time: 0.032846 seconds

Testing with array size: 100000
Insertion Sort time: 11.4047 seconds
Merge Sort time: 0.043985 seconds
```

三、程式邏輯

程式碼主要有 4 部分：Insertion sort、Merge sort、產生隨機 array 及主函式。

```
//insertion sort
void insertionSort(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 = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

(Insertion Sort)

```cpp
//merge sort
void merge(vector<int>& arr, int left, int right);
void mergeSort(vector<int>& arr) {
    merge(arr, 0, arr.size() - 1);
}

void merge(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge(arr, left, mid);
        merge(arr, mid + 1, right);

        int n1 = mid - left + 1;
        int n2 = right - mid;

        vector<int> L(n1), R(n2);

        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int i = 0; i < n2; i++)
            R[i] = arr[mid + 1 + i];

        int i = 0, j = 0, k = left;
        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++];
        }
    }
}
```

(Merge Sort)

```cpp
//access random array by using vector
vector<int> generateRandomArray(int size) {
    vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 1000000;
    }
    return arr;
}
```

(Generate random array)

```cpp
int main() {
    //define size of the array
    vector<int> sizes = {100,500,1000,2500, 5000,7500, 10000,25000 ,50000,75000, 100000};

    for (int size : sizes) {
        cout << " Testing with array size: " << size << endl;

        //generate the array we want to sort
        vector<int> arr1 = generateRandomArray(size);
        vector<int> arr2 = arr1;


        auto start = std::chrono::high_resolution_clock::now();
        insertionSort(arr1);
        auto end = std::chrono::high_resolution_clock::now();
        chrono::duration<double> insertionSortTime = end - start;
        //print time neede in insertion sort
        cout << " Insertion Sort time: " << insertionSortTime.count() << " seconds" << endl;

        start = std::chrono::high_resolution_clock::now();
        mergeSort(arr2);
        end = std::chrono::high_resolution_clock::now();
        chrono::duration<double> mergeSortTime = end - start;
        //print time neede in merge sort
        cout << " Merge Sort time: " << mergeSortTime.count() << " seconds" << endl;

        cout << endl;
    }

    return 0;
}
```

(主函式)

透過課本上的 Pseudo Code，我實作了兩種分類法的程式，並增加檢驗一些不同大小的 array 來測驗演算法的時間複雜度。從課堂上所學我們可知：

演算法	Best Case	Average	Worst
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$

Insertion Sort 的概念類似撲克牌的整理手牌，透過迭代進行插入並比較，從 index 小的到最後，因此需要進行 n 的平方次，時間複雜度等於$O(n^2)$。最好情況是代表不需要重複迭代，只需要 run 過 n 次就可以；但通常 array 不可能如此完美，除非是接近完全分好，否則在時間複雜度上還是偏高。因此 Insertion Sort 會比較適合小型數據集或幾乎已排序的資料，如：手動整理少量數據、整理幾乎分類好的數據。

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Merge Sort 的核心概念則為 Divide and Conquer，透過將一個完整 array 分成兩個小的 sub-array，再從兩個 sub-array 由 index 小而大互相比較，再回傳結果到原本的陣列。以上過程也不只執行一次，而是以類似遞迴的方式進行：

1. Divide：Merge Sort 會不斷地將 array 對半分割，直到每個 sub-array 只剩下一個元素。假設原本的 array 長度為 n，大概可以分割$\log_2 n$次($\frac{n}{2^k} = 1$，$solve\ for\ k$)。
2. Conquer：當所有 sub-array 拆分完成後，Merge Sort 會進行合併。因為需要掃過所有元素，因此時間複雜度是$O(n)$。

結合上述過程，可以知道 Merge Sort 時間複雜度約為

$O(n) \times O(\log_2 n) = O(n \log_2 n)$。因此 Merge Sort 適合較大規模數據或需要保證穩定性，如：排序大型檔案、大量沒有排序的資料，排序大量時幾乎都會比起 Insertion Sort 來的更快。

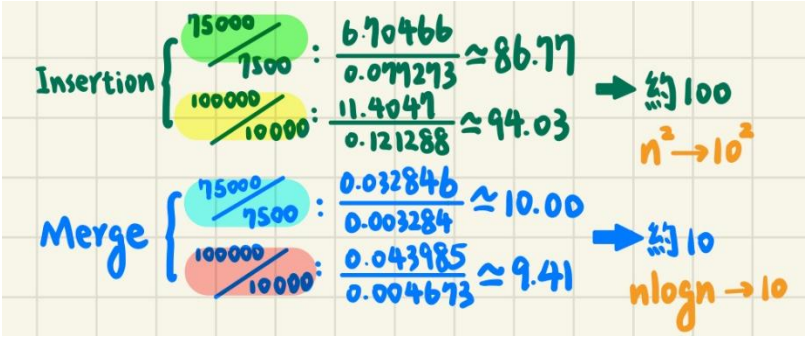
```
MERGE(A, p, q, r)
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

四、程式結果

array size	Insertion	Merge
2500	0.008201 s	0.001256 s
5000	0.0355 s	0.004063 s
7500	0.077273 s	0.003284 s
10000	0.121288 s	0.004673 s
25000	0.743966 s	0.009519 s
50000	2.96421 s	0.035724 s
75000	6.70466 s	0.032846 s
100000	11.4047 s	0.043985 s

以上為程式的 running time 時間整理，透過觀察可以發現：



比較各個放大十倍的 array size，可發現 Insertion Sort 和 Merge Sort 的時間複雜度確實約等於$O(n^2)$和$O(n \log_2 n)$。以上計算的會是程式的 sorting time，因為我將時間的起點和終點放在 sort 函式前後，因此會單純計算函式的 sorting time。

```
auto start = std::chrono::high_resolution_clock::now();
insertionSort(arr1);
auto end = std::chrono::high resolution clock::now();
```

五、結論

透過本次作業的討論，我發現課堂上的理論可以透過簡單的方法來驗證，也讓這些看似抽象的概念變的立體。藉此我也更了解程式的時間複雜度概念，例如：在實現上，雖然 Insertion Sort 演算法的行數短很多，但效能上卻遠不如 Merge Sort，且概念更類似遞迴的 Merge Sort 也不會因此增加大量的執行時間。

教授在講到其他分類法時我也在腦海中試著想像了複雜度與行數的關係，藉此我發現往往節省行數的意義在於增加減少判斷與遞迴，但站在時間複雜度的角度上，不停新增插入改變的處理方法(Insertion Sort)雖然邏輯簡單，卻增加了太多重複的邏輯過程。因此最終結果雖然簡潔卻不甚理想，反而透過多層遞迴(Merge Sort)的效果還可能更好。這些收穫能讓我在下次進行類似運算、撰寫、或判斷時，邏輯更加清晰，並根據不同需求，例如可讀性或省效能，來調配出最適合的方法或演算法。