資料結構 (Data Structure) 作業一

姓名：謝政彥
學號：109753207

一、 作業目標：比較 insertion sort，merge sort， randomized quick sort，counting sort，與任一個其他時間複雜度為 O(nlogn) 【本作業選擇 heap sort】的排序演算法

二、 比較方法：

(一) Input array：

1. 大小：2^k (k=10, 11, 12, ……., 30)
2. 種類：
   (1) Uniformly Randomly
   (2) Almost Sorted
3. 數量：每種類按不同大小，各產出 10 個 array

(二)執行時間：各演算法針對上述 10 個 array 排序所需時間的平均

(三)折線圖：針對上述 array 種類，分別產生一個折線圖(包含 5 條演算法的折線)，x 軸為陣列大小，y 軸為執行時間

三、 實驗環境：

(一)電腦：Aspire M1935

1. 處理器： Intel(R) Core(TM)i5-3350U CPU @3.100GHz 3.10GHz
2. 記憶體(RAM)： 16.0GB
3. 系統類型： 64 位元作業系統，x64 型處理器
4. Windows 規格
   (1) 版本 Windows 10 家用版
   (2) 版本 2004

(二)C++開發工具

整合開發環境(Integrated Development Environment: IDE)：Visual Studio 2019

四、 演算法程式來源：詳第 8 頁。

五、 實驗程式碼：詳第 9 頁。

六、 實驗過程及結果：

(一)程式設計原則是，以 2 的冪次，即 10 至 30 做外圍迴圈，接著 1 至 10 作為重新產生陣列的迴圈，再來是 1 至 5 執行 5 種排序演算法的內層迴圈，內層迴圈運用已排序完成之陣列，隨機抽取 100 個位置，並替換 1-1000 隨機產生數字的陣列，做為執行接近排序完成(Almost Sorted)陣列排序演算。各演算法針對上述 10

個陣列排序所需時間的平均值，以 CSV 格式輸出。

(二)原始數據如表 1 (詳第 7 頁)，其中 InsertionSort 在均勻隨機(Uniformly Randomly)陣列執行到 2^21 大小時，平均時間已達約 46 分鐘，即暫停後續執行，並按其在 EXCEL 趨勢圖公式(y =(9)*(10^(-7))*(X^1.9747)，另詳圖 1)，推估相關數據。

(三)Randomized QuickSort 原參用自

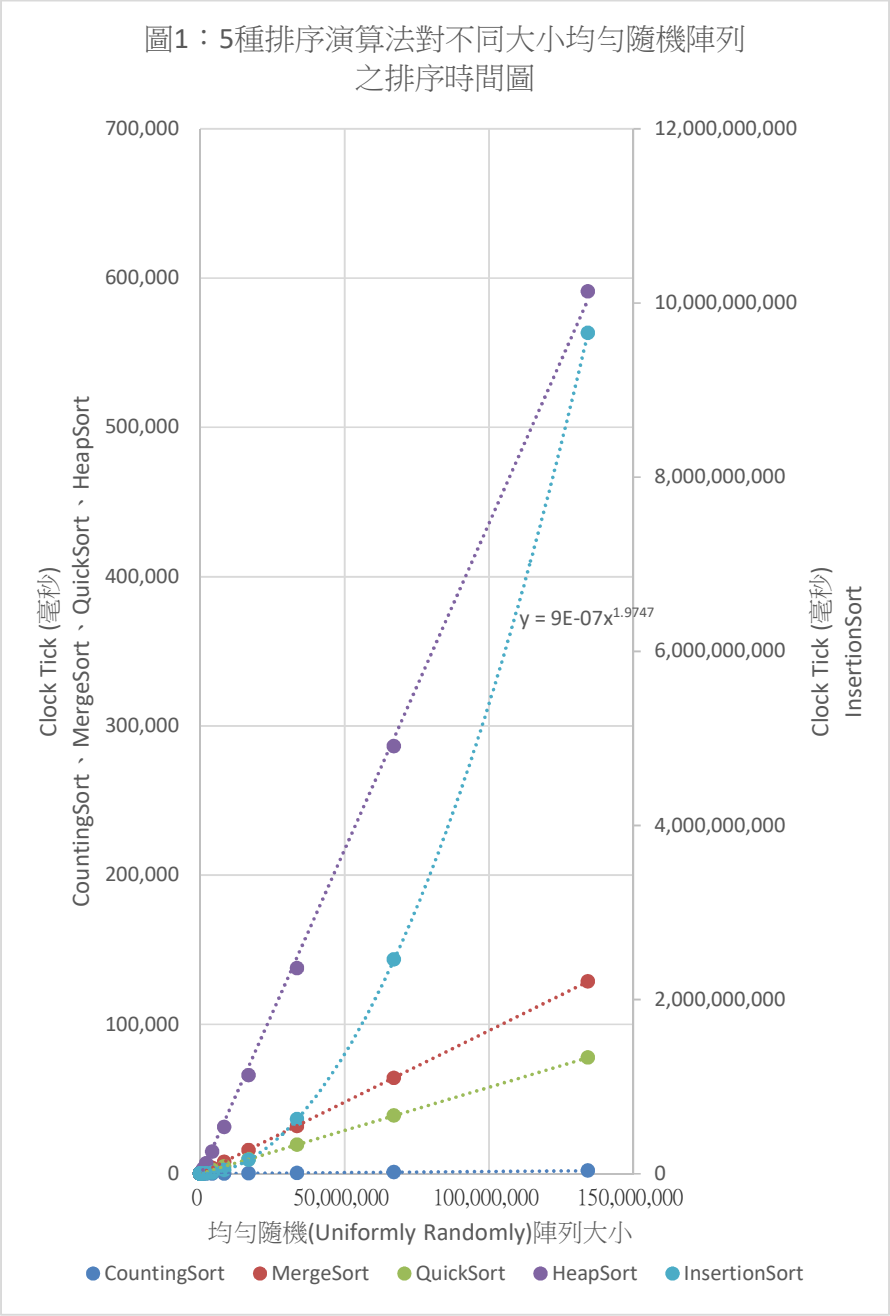
「 https://www.sanfoundry.com/cpp-program-implement-quick-sort-using-randomisation/ 」 取得的程式碼，但執行到 2^22 大小時即中止執行，並出現代碼 1073741571，在試著解決問題同時，先將 GeeksforGeeks 取得屬於解決「Dutch Flag Problem」的 3-way 程式，因沒有 randomized pivot 部分，自行加上前述程式碼中 randomized pivot 的片段(請參閱第 18、19 頁)，重新再跑又可運行，於是**全部程式自 2^10 開始重跑**，包括第(一)點部分。

(四)所有程式除了自 2^22 開始停止執行均勻隨機(Uniformly Randomly)陣列的 InsertionSort 外，接著均可以完成到 2^26，自 2^27 開始出現代碼 1073741855，並停止執行。經參考註 1 有關 STACK (堆疊配置)相關網路資訊，在 Visual Studio 2019 開發環境中設定連結器選項：1.開啟專案的 [屬性頁] 對話方塊、2.選取 連結器資料夾、3.選取 系統屬性頁、4.修改下列屬性之一：•堆疊基本配置大小、• Stack Reserve Size，筆者並未修改其建議屬性，而是直接啟用大型記憶體後，2^27 大小陣列即可用演算法排序。

(五)惟執行到 2^28 大小時，再次出現代碼 1073741855，並停止執行，經檢視其停止位置，發現連 2^28 大小陣列都無法初始化，因原本程式設計這樣大小陣列計有 3 個，包括儲存保留原始 1-1000 隨機產生的陣列、隨機抽取 100 個位置並替換 1-1000 隨機數字的陣列，以及複製前述陣列給演算法執行得陣列，經取消 Almost Sorted 陣列的排序指令，並註解相關陣列後，也只有 Randomized QuickSort 及 HeapSort 在既有的條件下可以繼續運行，但執行至 2^29 就再也無法執行。

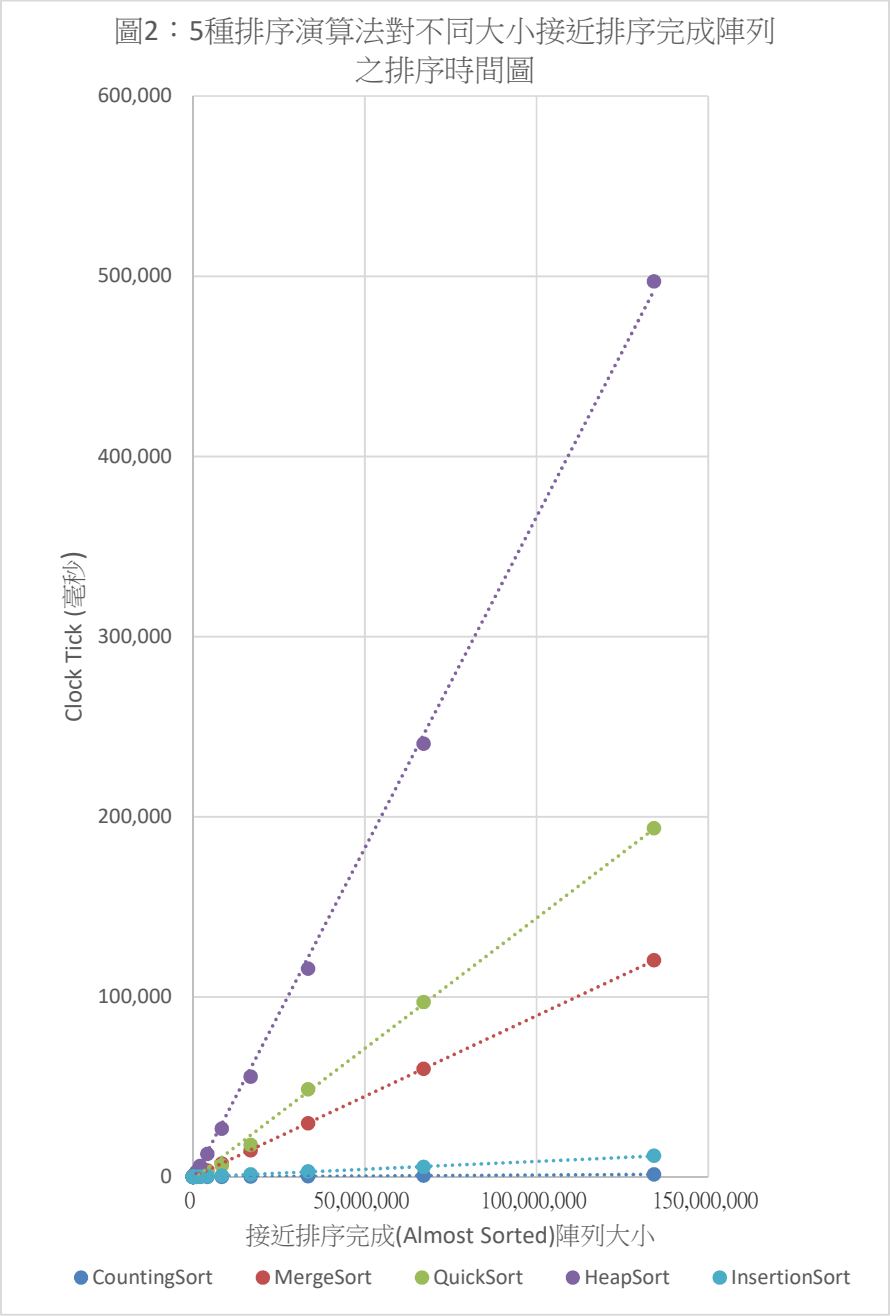
(六)以下謹就所獲得的執行成果，繪製相關折線圖，如圖 1、圖 2，說明如下：

1. 均勻隨機(Uniformly Randomly)陣列



在均勻隨機(Uniformly Randomly)陣列的圖 1 中，可以清楚看到 InsertionSort 排序時間最長 (按：執行時間請看右邊標示)，符合其平均 O(n^2)的時間複雜度，如圖中趨勢圖公式所示，X (即陣列大小) 的冪次為 1.9747，非常接近 2。在這個實驗中，演算法執行最佳的是 CountingSort ，這也符合其時間複雜度 O(n+k)，因為 K 最大只到 1,000，遠低於陣列大小 n，所以其趨勢圖基本上是線性關係。其他 3 種演算法平均時間複雜度都是 O(nlogn)，屬線性對數（準線性）時間關係(註 2)，其執行排序所需時間以 Randomized QuickSort (3-Way)最快，其次 MergeSort，最後是 HeapSort。

2. 接近排序完成(Almost Sorted)陣列



在接近排序完成(Almost Sorted)的陣列，在排序上一般是屬於比較好的狀況，即所謂 Best case，可以很清楚從圖 1 及圖 2 中比較發現，InsertionSort 執行時間大幅縮短，只略輸 CountingSort (按：時間複雜度 O(n+k))一籌，這也符合其 Best case 是 O(n) 的時間複雜度。其他 3 種演算法，在 Best case 時均是 O(nlogn)，從圖 2 中看到，這次 MergeSort 比 Randomized QuickSort (3-Way)來的快，HeapSort 仍然墊底。

七、 心得、疑問及遇到困難

(一)心得

透過這次作業，深刻體驗相同的資料結構，不同的演算法，執行時間的差異極大，如執行到 2^21 大小之均勻隨機(Uniformly Randomly)陣列時，InsertionSort (按：平均時間複雜度 O(n^2)) 需要約 46 分鐘，而 CountingSort (按：平均時間複雜度 O(n+k)) 用不到 1 秒。此外，即使是相同的演算法，對不同的資料結構，其演算能力也可能不同，最明顯的也是 InsertionSort，對接近排序完成(Almost Sorted)陣列，以相同 2^21 大小為例，執行時間大幅縮短，時間複雜度只有 O(n)，遠優於 MergeSort、Randomized QuickSort (3-Way)及 HeapSort (按：三者 best case 時間複雜度均為 O(nlog(n))。因此，清楚資料結構，善選適用之演算法才是上策。

此外，執行過程發生錯誤訊息，也強迫自己去蒐集資料，解決問題，也重新檢視程式執行記憶體的配置，如 stack、heap 等 (註 3，也還沒真的弄懂)。我想心得是，發現自己有很多不足的地方。

(二)疑問

1. 錯誤訊號「出現代碼 1073741571」

   第一次遇到比較嚴重的錯誤訊號「出現代碼 1073741571」，是在執行 Randomized QuickSort 時，當陣列大小到了 2^21 時，就跳出前述錯誤訊號，並中止執行程序。當時並沒有真正解決問題，只是試著將從 GeeksforGeeks 找到屬於解決「Dutch Flag Problem」的 3-way 程式 (一開始有試跑，但沒有 randomized pivot 又被擱下)，再自行加上原取自

   「 https://www.sanfoundry.com/cpp-program-implement-quick-sort-using-randomisation/ 」程式碼中的 randomized pivot 片段，既然可以運行，就用整合版本執行程式。

2. 錯誤訊號「出現代碼 1073741855」

   接著在執行到 2^27 大小時，出現代碼 1073741855，並停止執行，經參考有關 STACK (堆疊配置)等相關網路資訊，調整 Visual Studio 2019 開發環境中設定連結器選項 (專案→屬性→連結器→系統→啟用大型記憶體)後，2^27 大小陣列即可用演算法排序，只是到了 2^28 又卡關。

3. 疑問是，如何才是真正解決的方法？

(三)遇到困難

此次作業遇到的困難是出現錯誤訊息時，不知採取的策略是否妥適，雖然網路上有很多資訊，但執行程式需要的時間很長，因此消化資訊及測試的過程顯得有點不足。其他如測試演算法 (包括排序前後是否正確、陣列的規劃)、熊熊發現程式運作被電腦休眠打斷，或運用輸出 CSV 檔方式，讓自己不用守在電腦旁等，也就微不足道了。

八、 參考資料

註 1：/STACK (堆疊配置)
https://docs.microsoft.com/zh-tw/cpp/build/reference/stack-stack-allocations?view=vs-2019

註 2：時間複雜度
https://zh.wikipedia.org/wiki/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6

註 3：Memory Layout of C Programs
https://www.geeksforgeeks.org/memory-layout-of-c-program/

✧ 原始數據

單位：Clock Tick(毫秒)

陣列		CountingSort		MergeSort		QuickSort		HeapSort		InsertionSort[1]	
大小	2 的冪次	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd
1,024	10	0	0	1	1	1	0	2	2	1	0
2,048	11	0	0	1	1	1	1	3	3	3	0
4,096	12	0	0	3	3	2	1	7	7	10	0
8,192	13	0	0	7	6	5	3	16	16	42	0
16,384	14	0	0	15	13	10	6	35	35	168	1
32,768	15	0	0	29	27	20	11	75	72	676	3
65,536	16	1	0	58	55	39	22	159	149	2,714	6
131,072	17	1	1	117	110	77	42	340	310	10,736	11
262,144	18	3	3	237	221	153	88	722	653	42,992	23
524,288	19	5	6	473	443	305	179	1,534	1,377	171,980	45
1,048,576	20	12	11	954	888	607	397	3,261	2,901	687,475	92
2,097,152	21	25	23	1,919	1,792	1,214	923	6,967	6,106	2,758,136	189
4,194,304	22	52	43	3,894	3,628	2,439	2,338	14,847	12,772	10,378,023	364
8,388,608	23	115	86	7,838	7,316	4,853	6,437	31,360	26,678	40,722,656	743
16,777,216	24	233	177	15,860	14,816	9,755	17,676	65,910	55,666	159,792,930	1,427
33,554,432	25	480	339	31,856	29,763	19,456	48,568	137,611	115,587	627,016,574	2,970
67,108,864	26	978	677	64,169	59,927	38,998	97,011	286,354	240,486	2,460,370,330	5,635
134,217,728	27	1,963	1,406	128,807	120,238	77,775	193,530	591,014	497,079	9,654,325,597	11,768
268,435,456	28					156084		1230314			
536,870,912	29										
1,073,741,824	30										

[1] InsertionSort 在均勻隨機(Uniformly Randomly)陣列執行到 2^21 大小時，平均時間已達約 46 分鐘，即暫停後續執行，並按其在 EXCEL 趨勢圖公式(y = 9*10^(-7)*(X^1.9747)推估後面數字(如藍底)。

✧ 演算法程式來源

序號	演算法	程式來源	備註
1	CountingSort	https://www.programiz.com/dsa/counting-sort	Best case: O(n+k), Average case: O(n+k), Worst case: O(n+k)
2	MergeSort	GeeksforGeeks	Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn)
3	Randomized QuickSort	GeeksforGeeks 加上 https://www.sanfoundry.com/cpp-program-implement-quick-sort-using-randomisation/ 的隨機取 pivot	Best case: O(nlogn), Average case: O(nlogn), Worst case: O(n^2)
4	HeapSort	GeeksforGeeks	Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn)
5	InsertionSort	GeeksforGeeks	Best case: O(n), Average case: O(n^2), Worst case: O(n^2)

✧ 實驗程式碼

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <random>
#include <fstream>
#include <string>

using namespace std;

/****** CountingSort --> Code Source:    https://www.programiz.com/dsa/counting-sort ******/
/****** Best case: O(n+k), Average case: O(n+k), Worst case: O(n+k) ******/
void countSort(int* arr, int size, int max);

/****** MergeSort --> Code Source: GeeksforGeeks ******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn) ******/
void merge(int* arr, int l, int m, int r);
void mergeSort(int* arr, int l, int r);

/****** Randomized QuickSort --> Code Source: GeeksforGeeks  加上
https://www.sanfoundry.com/cpp-program-implement-quick-sort-using-randomisation/  的隨機取 pivot
******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(n^2) ******/
void swap(int* a, int* b);
void partition(int a[], int l, int r, int& i, int& j);
void quicksort(int a[], int l, int r);

/****** HeapSort --> Code Source: GeeksforGeeks ******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn) ******/
void heapify(int* arr, int n, int i);
void heapSort(int* arr, int n);

/****** InsertionSort --> Code Source: GeeksforGeeks ******/
/****** Best case: O(n), Average case: O(n^2), Worst case: O(n^2) ******/
void insertionSort(int* arr, int n);

/* Function to print an array */
```

```
void printArray(int* A, int size);

int main(int argc, char* argv[])
{
	int mySum[21][5][2] = { 0 };	//排序時間加總[冪次 10-30][演算法 1-5][隨機取號、排序後抽換 100 個數字]
	int myAvg[21][5][2] = { 0 };	//平均排序時間[冪次 10-30][演算法 1-5][隨機取號、排序後抽換 100 個數字]

	for (int myPower = 10; myPower <= 30; myPower++) {		//陣列數 2^10-->2^30 迴圈

		for (int myRepeat = 1; myRepeat <= 10; myRepeat++) {	//重複次數1-->10 迴圈，每次產生不同陣列

			//隨機建立整數陣列
			int n = (int)(pow(2, myPower));		//陣列數
			int* oriArr = new int[n];		//for 隨機取號(1-1000)之原始陣列
			int* oriArr2 = new int[n];		//for 排序號再抽 100 個序位置換(1-1000)亂數之原始陣列 2
			int* arr = new int[n];			//複製原始陣列，讓演算法都用同一陣列

			//檢視
			cout << "***************myPower=" << myPower << endl;
			cout << "***************myRepeat=" << myRepeat << endl;

			random_device rd;
			/*  梅森旋轉演算法  */
			mt19937 generator(rd());
			uniform_int_distribution<int> unif(1, 1000);
			for (int i = 0; i < n; i++) {
				oriArr[i] = (int)(unif(generator));			//隨機取號(1-1000)
				//cout << oriArr[i] << "->";				//檢視
			}
			//cout << endl;

			//依序跑 5 種排序演算法
			for (int sortAlgo_no = 1; sortAlgo_no <= 5; sortAlgo_no++) {	//5 種演算法迴圈

				//複製原始陣列 oriArr[]到 arr[]以確保各種演算法所用陣列相同
				for (int i = 0; i < n; i++) {
					arr[i] = oriArr[i];
				}
```

```
//排序&計時
clock_t start1, end1;
start1 = clock();

switch (sortAlgo_no) {
case 1:
	cout << "CountingSort" << endl;
	countSort(arr, n, 1000);
	break;
case 2:
	cout << "MergeSort" << endl;
	mergeSort(arr, 0, n - 1);
	break;
case 3:
	cout << "QuickSort" << endl;
	quicksort(arr, 0, n - 1);
	break;
case 4:
	cout << "HeapSort" << endl;
	heapSort(arr, n);
	break;
case 5:
	cout << "InsertionSort" << endl;
	insertionSort(arr, n);
	break;
}

end1 = clock();
mySum[myPower - 10][sortAlgo_no - 1][0] += (end1 - start1);	//排序時間加總
cout << (end1 - start1) << " Ticks for 1st sorting" << endl;

//檢視
//cout << "After:" << endl;
//printArray(arr, n);

//隨機從排序好陣列抽取 100 個序位，並隨機替換數字(1-1000)
uniform_int_distribution<int> unif2(0, n - 1);
if (sortAlgo_no == 1) {
	for (int j = 0; j < 100; j++) {
```

```
                int RanSn = (int)(unif2(generator));            //隨機序位
                int ChangeNum = (int)(unif(generator));         //隨機數字(1-1000)
                arr[RanSn] = ChangeNum;
            }
            //複製抽換過的陣列到 oriArr2[]
            for (int i = 0; i < n; i++) {
                oriArr2[i] = arr[i];
            }
        }

        //第 1 個演算法用的 arr[]是抽換過的，其他則複製抽換過且未排序的陣列 oriArr2[]到 arr[]以確保各種演算法所用陣列相同
        if (sortAlgo_no != 1) {
            for (int i = 0; i < n; i++) {
                arr[i] = oriArr2[i];
            }
        }

        //從新排序計時
        clock_t start2, end2;
        start2 = clock();

        switch (sortAlgo_no) {
        case 1:
            cout << "CountingSort" << endl;
            countSort(arr, n, 1000);
            break;
        case 2:
            cout << "MergeSort" << endl;
            mergeSort(arr, 0, n - 1);
            break;
        case 3:
            cout << "QuickSort" << endl;
            quicksort(arr, 0, n - 1);
            break;
        case 4:
            cout << "HeapSort" << endl;
            heapSort(arr, n);
            break;
        case 5:
```

```
                cout << "InsertionSort" << endl;
                insertionSort(arr, n);
                break;
            }

            end2 = clock();
            mySum[myPower - 10][sortAlgo_no - 1][1] += (end2 - start2);
            cout << (end2 - start2) << " Ticks for 2nd sorting" << endl;

            //檢視
            //cout << "After2:" << endl;
            //printArray(arr, n);
        }
        delete[] arr;
        delete[] oriArr2;
        delete[] oriArr;
        cout << "Already delete arr" << endl;
    }
    cout << endl;

    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 2; j++) {
            myAvg[myPower - 10][i][j] = mySum[myPower - 10][i][j] / 10;
        }
    }
    //平均執行時間輸出到 CSV 檔
    ofstream myfile;
    string filename = "myPower_k2_" + to_string(myPower) + ".csv";
    myfile.open(filename);
    myfile << ",CountingSort,,MergeSort,,QuickSort,,HeapSort,,InsertionSort\n";
    myfile << "myPower, 1st Round, 2nd Round, 1st Round, 2nd Round, 1st Round, 2nd Round, 1st
Round, 2nd Round, 1st Round, 2nd Round\n";
    myfile << myPower;
    myfile << ",";
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 2; j++) {
            myfile << myAvg[myPower - 10][i][j];
            myfile << ",";
        }
```

```
            }
            myfile << "\n";
            myfile.close();
        }

        cout << "Done!" << endl;
        system("PAUSE");
        return 0;
}


/****** MergeSort --> Code Source: GeeksforGeeks ******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn) ******/
// Merges two subarrays of arr[]
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]

void merge(int* arr, int l, int m, int r)
{
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        /* create temp arrays */
        //int L[n1], int R[n2];
        int* L = new int[n1];
        int* R = new int[n2];

        /* Copy data to temp arrays L[] and R[] */
        for (i = 0; i < n1; i++)
                L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
                R[j] = arr[m + 1 + j];

        /* Merge the temp arrays back into arr[l..r]*/
        i = 0; // Initial index of first subarray
        j = 0; // Initial index of second subarray
        k = l; // Initial index of merged subarray
        while (i < n1 && j < n2) {
```

```
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy the remaining elements of L[], if there
        are any */
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy the remaining elements of R[], if there
        are any */
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }

        delete[] R;
        delete[] L;
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
//void mergeSort(int arr[], int l, int r)
void mergeSort(int* arr, int l, int r)
{
        if (l < r) {
            // Same as (l+r)/2, but avoids overflow for
            // large l and h
```

```
            int m = l + (r - l) / 2;

            // Sort first and second halves
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

            merge(arr, l, m, r);
        }
}

/****** Randomized QuickSort --> Code Source:
https://www.sanfoundry.com/cpp-program-implement-quick-sort-using-randomisation/ ******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(n^2) ******/
// Swapping two values.
void swap(int* a, int* b)
{
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
}

void partition(int a[], int l, int r, int& i, int& j)
{
        i = l - 1, j = r;
        int p = l - 1, q = r;
        int v = a[r];

        while (true) {
            // From left, find the first element greater than
            // or equal to v. This loop will definitely
            // terminate as v is last element
            while (a[++i] < v)
                ;

            // From right, find the first element smaller than
            // or equal to v
            while (v < a[--j])
                if (j == l)
```

```
                    break;

        // If i and j cross, then we are done
        if (i >= j)
                break;

        // Swap, so that smaller goes on left greater goes
        // on right
        swap(a[i], a[j]);

        // Move all same left occurrence of pivot to
        // beginning of array and keep count using p
        if (a[i] == v) {
                p++;
                swap(a[p], a[i]);
        }

        // Move all same right occurrence of pivot to end of
        // array and keep count using q
        if (a[j] == v) {
                q--;
                swap(a[j], a[q]);
        }
}

// Move pivot element to its correct index
swap(a[i], a[r]);

// Move all left same occurrences from beginning
// to adjacent to arr[i]
j = i - 1;
for (int k = l; k < p; k++, j--)
        swap(a[k], a[j]);

// Move all right same occurrences from end
// to adjacent to arr[i]
i = i + 1;
for (int k = r - 1; k > q; k--, i++)
        swap(a[i], a[k]);
```

```
}

// 3-way partition based quick sort
void quicksort(int a[], int l, int r)
{
	if (r <= l)
		return;

	int i, j;

	// Random selection of pivot.
	int pvt, n;
	n = rand();
	// Randomizing the pivot value in the given subpart of array.
	pvt = l + n % (r - l + 1);
	// Swapping pvt value from high, so pvt value will be taken as pivot while partitioning.
	swap(&a[r], &a[pvt]);


	// Note that i and j are passed as reference
	partition(a, l, r, i, j);

	// Recur
	quicksort(a, l, j);
	quicksort(a, i, r);
}

/*****************************************************************************/
// Partitioning the array on the basis of values at high as pivot value.
/*
int Partition(int* a, int low, int high)
{
	int pivot, index, i;
	index = low;
	pivot = high;

	// Getting index of pivot.
	for (i = low; i < high; i++)
	{
```

```
            if (a[i] < a[pivot])
            {
                //cout << "Partition --> SWAPING" << endl;
                swap(&a[i], &a[index]);
                index++;
            }

        }
        // Swapping value at high and at the index obtained.
        swap(&a[pivot], &a[index]);
        //cout << "DONE_Partition" << endl;
        return index;
}
*/
// Random selection of pivot.
/*
int RandomPivotPartition(int* a, int low, int high)
{
        int pvt, n;
        n = rand();
        // Randomizing the pivot value in the given subpart of array.
        pvt = low + n % (high - low + 1);

        // Swapping pvt value from high, so pvt value will be taken as pivot while partitioning.
        swap(&a[high], &a[pvt]);
        return Partition(a, low, high);
}
*/
// Implementing QuickSort algorithm.
/*
int QuickSort(int* a, int low, int high)
{
        int pindex;
        if (low < high)
        {
            // Partitioning array using randomized pivot.
            pindex = RandomPivotPartition(a, low, high);
            // Recursively implementing QuickSort.
            QuickSort(a, low, pindex - 1);
```

```
            QuickSort(a, pindex + 1, high);
        }
        return 0;
}
*/


    /****** InsertionSort --> Code Source: GeeksforGeeks ******/
    /****** Best case: O(n), Average case: O(n^2), Worst case: O(n^2) ******/
void insertionSort(int* arr, int n)
{
        int i, key, j;
        for (i = 1; i < n; i++)
        {
                key = arr[i];
                j = i - 1;

                /* Move elements of arr[0..i-1], that are
                greater than key, to one position ahead
                of their current position */
                while (j >= 0 && arr[j] > key)
                {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = key;
        }
}


/****** CountingSort --> https://www.programiz.com/dsa/counting-sort ******/
/****** Best case: O(n+k), Average case: O(n+k), Worst case: O(n+k) ******/
void countSort(int* arr, int size, int max) {
        // The size of count must be at least the (max+1) but
        // we cannot assign declare it as int count(max+1) in C++ as
        // it does not support dynamic memory allocation.
        // So, its size is provided statically.
        int* output = new int[size + 1];
        int* count = new int[max + 1]{ 0 };
```

```
//int max = arr[0];

// Find the largest element of the array
/*
for (int i = 1; i < size; i++) {
        if (arr[i] > max)
                max = arr[i];
}

// Initialize count array with all zeros.
for (int i = 0; i <= max; ++i) {
        count[i] = 0;
}
*/

// Store the count of each element
for (int i = 0; i < size; i++) {
        count[arr[i]]++;
}

// Store the cummulative count of each array
for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
}

// Find the index of each element of the original array in count array, and
// place the elements in output array
for (int i = size - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
}

// Copy the sorted elements into original array
for (int i = 0; i < size; i++) {
        arr[i] = output[i];
}

delete[] count;
delete[] output;
```

```
}


/****** HeapSort --> Code Source: GeeksforGeeks ******/
/****** Best case: O(nlogn), Average case: O(nlogn), Worst case: O(nlogn) ******/
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int* arr, int n, int i)
{
	int largest = i; // Initialize largest as root
	int l = 2 * i + 1; // left = 2*i + 1
	int r = 2 * i + 2; // right = 2*i + 2

	// If left child is larger than root
	if (l < n && arr[l] > arr[largest])
		largest = l;

	// If right child is larger than largest so far
	if (r < n && arr[r] > arr[largest])
		largest = r;

	// If largest is not root
	if (largest != i)
	{
		swap(arr[i], arr[largest]);

		// Recursively heapify the affected sub-tree
		heapify(arr, n, largest);
	}
}

// main function to do heap sort
void heapSort(int* arr, int n)
{
	// Build heap (rearrange array)
	for (int i = n / 2 - 1; i >= 0; i--)
		heapify(arr, n, i);

	// One by one extract an element from heap
```

```
	for (int i = n - 1; i > 0; i--)
	{
		// Move current root to end
		swap(arr[0], arr[i]);

		// call max heapify on the reduced heap
		heapify(arr, i, 0);
	}
}


/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int* A, int size)
{
	int i;
	for (i = 0; i < size; i++)
		printf("%d ", A[i]);
	printf("\n");
}
```