研究報告

報告撰寫者

學號：110704039

姓名：許甄芸

研究主題（樣本）

在 C++程式語言中，sorting 的方式有很多種，包含習題在內課本介紹了：

- Insertion sort
- Selection sort
- Bubble sort
- Quick sort(recursion 版本)

本研究主題旨在探討此四種撰寫方式對程式執行效率（時間）的影響。

研究方法

1. 撰寫程式量測每一個撰寫方式的執行時間，並加以比較。
2. 基於比較的公平性與正確性，四種撰寫方式的「測資」與「執行次數」必須相同。因此在執行上除了上述四種不同 sorting 方式外，需額外執行「執行次數」的「比較」。此一動作可能會影響比較結果，如後述。

測試資料來源或產生方式

為了確保四種執行方法的測資皆相同，使用<cstdlib>函式庫中的 rand 函數並固定亂數種子，就能達成目的。

```
#include<cstdlib>
……
srand(7);
int a = rand(), b = rand(), c = rand(), d = rand(), e = rand();
int array[5] = {a, b, c, d, e};
```

方法實作

Insertion sort; 實作程式如附錄一：Insertion sort.cpp

Selection sort; 實作程式如附錄二：Selection sort.cpp

Bubble sort; 實作程式如附錄三：Bubble sort.cpp

Quick sort(recursion 版本); 實作程式如附錄三：Quick sort.cpp

為量測到有效執行時間，必須重複執行函式數次。定義四個程式測試運算單位為公差為 5 的等差數列：

```
for(int m = 5; m <= 100; m+=5){
  for(int n = 1; n <= m; n++)
      {
          ……
      }
}
```

固定陣列長度為 5，分別執行 5、10、15…100 次：

```
for(int m = 5; m <= 100; m+=5){
  for(int n = 1; n <= m; n++)
      {
          int a = rand(), b = rand(), c = rand(), d =
          rand(), e = rand();
          int array[5] = {a, b, c, d, e};
      }
}
```

於每回合前後，計算 CPU 執行每回合的 clock tick 數目，並換算成執行時間輸出：

```
clock_t clicks = clock();
for(int m = 5; m <= 100; m+=5){
  for(int n = 1; n <= m; n++)
      {
          ……
      }
}
clicks = clock() - clicks;
cout << "The sorting time: " << (float)clicks /
CLOCKS_PER_SEC << endl;
```

執行平台

作業系統版本與硬體規格如下：

CPU type	AMD Ryzen 7 4700U with Radeon Graphics 2.00 GHz
Memory size	16.0GB
Kernel version	Windows 11 家用版
C version	Dev-C++
Machine type	64 位元作業系統，x64 型處理器

實作結果與分析

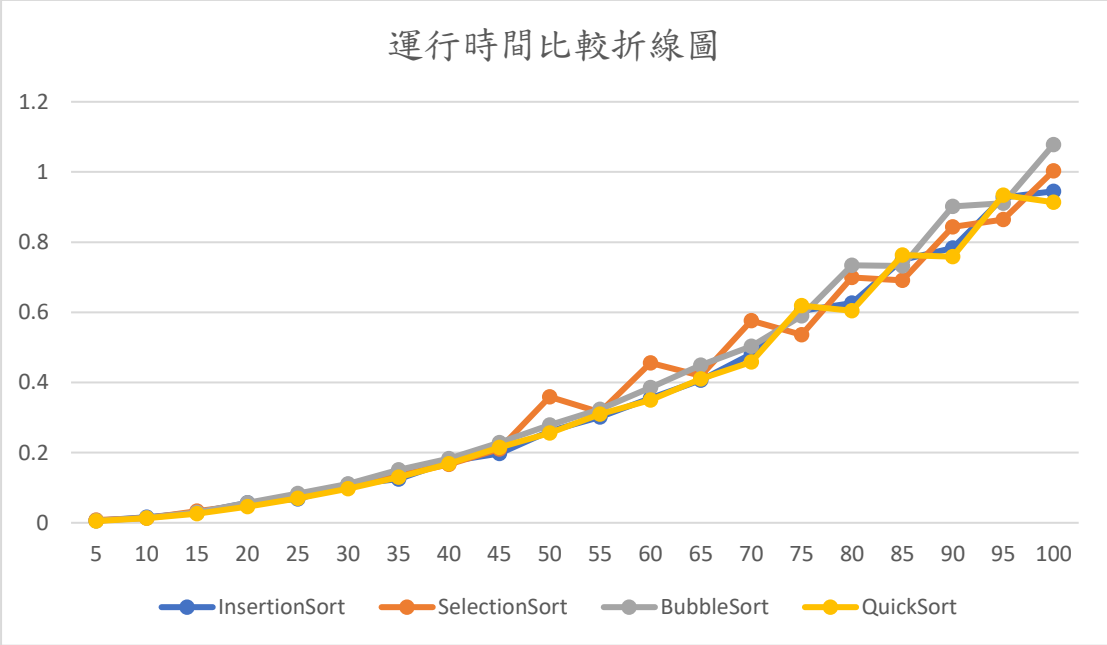
實作結果如下圖所示，其中

- 四個程式測試運算單位為公差為 5 的等差數列:5、10、15…100，如下圖x軸數字標示。
- 量測的時間單位為秒，如下圖y軸數字標示。
- 藍色曲線為”Insertion sort ”測試版本，橘色曲線為”Selection sort ”測試版本，灰色曲線為”Bubble sort ”測試版本，黃色曲線為”Quick

sort ” 測試版本。

由下圖中我們可以看出四種撰寫方式的執行時間一開始都非常接近，隨著執行次數的增加，折線圖開始變得崎嶇，但仔細觀察可以發現藍色的 Insertion sort 和灰色的 Bubble sort 相較之下是較穩定的，詳細資料可參考以下附圖。同時可以推測在此測資設定下，四種方式對程式執行效率（時間）的影響不大。

演算法	時間複雜度			空間複雜度	穩定性	類型
	Best	Worst	Avg			
選擇排序法(Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不穩定	選擇
插入排序法(Insertion Sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)	穩定	插入
氣泡排序法(Bubble Sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)	穩定	交換
謝爾排序法(Shell Sort)	O(n)	$O(n^2)$~ $O(n^{1.5})$	$O(n^{5/4})$	O(n) + O(1)	不穩定	插入
搖晃排序法(Shaker Sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)	穩定	交換
快速排序法(Quick Sort)	O(n log n)	$O(n^2)$	O(n log n)	O(log n)~O(n)	不穩定	交換
合併排序法(Merge Sort)	O(n log n)	O(n log n)	O(n log n)	O(n)	穩定	合併
堆積排序法(Heap Sort)	O(n log n)	O(n log n)	O(n log n)	O(n) + O(1)	不穩定	選擇
基數排序(Radix Sort)	O(d×(n+r))	O(d×(n+r))	O(d×(n+r))	O(n×r)	穩定	分配



問題與討論

1. 測試資料的來源或產生方式。

為了確保四種執行方法的測資皆相同，使用<cstdlib>函式庫中的 rand 函數並固定亂數種子，就能達成目的。

```
#include<cstdlib>
……
```

```
srand(7);
int a = rand(), b = rand(), c = rand(), d = rand(), e
= rand();
int array[5] = {a, b, c, d, e};
```

2. 測試資料元素個數（陣列大小）以下以 n 表示。

以$n = 5$為例，$\{1,2,3,4,5\}$ 與 $\{5,4,3,2,1\}$ 與 $\{3,2,5,1,4\}$ 相同的方法因測試資料原始排列不同，其執行的時間也會不同。所以陣列大小n相同的測試資料不能只測一、二筆，需要測很多筆可能（隨機）的原始排列，再求排序的平均執行時間，以以下程式為例:

(詳細實作程式請參考附錄: Q2_InsertionSort.cpp、Q2_SelectionSort.cpp、Q2_BubbleSort、Q2_QuickSort)

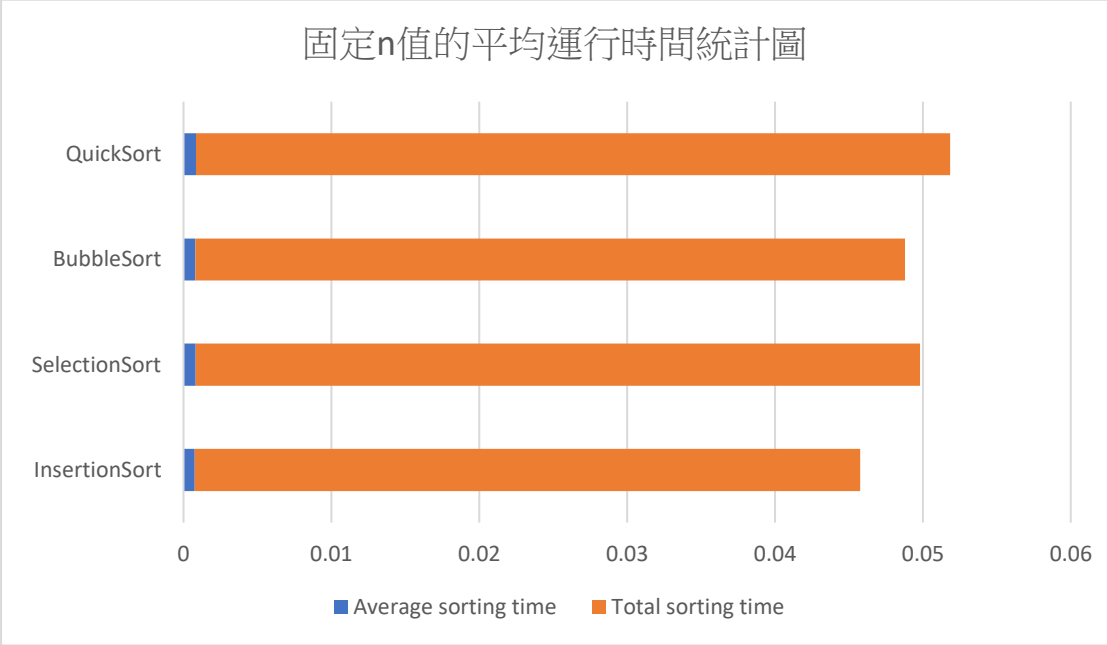
用<algorithm>函式庫中的 random_shuffle 函數打亂 array 中的 1~5 元素

```
#include<algorithm>
……
int array[5] = {1, 2, 3, 4, 5};
random_shuffle(array, array + 5);
```

由於 5 階乘的結果為 60，讓迴圈執行 60 次再計算運行時間之平均較為準確

```
for(int j = 0; j < 60; j++)
  {
      random_shuffle(array, array + 5);
      ……
  }
cout << "The average sorting time: " << ((float)clicks
/ CLOCKS_PER_SEC) / 60 << endl;
```

由程式實作的測試結果如下



實測結果	Average sorting time	Total sorting time
Insertion sort	0.00075s	0.045s
Selection sort	0.000816667s	0.049s
Bubble sort	0.0008s	0.048s
Quick sort	0.00085s	0.051s

3. 基於比較的公平性，如何確保不同的方法採用相同的測試資料。

若以固定n值的第 2 題來舉例，一開始設定$array = \{1,2,3,4,5\}$，接著使用$random_shuffle$打亂陣列的順序，這樣便確保測資中不會出現已排序的資料，也就是陣列原始的排序:1,2,3,4,5。而如果要確保每筆資料皆不會重複，可以宣告一個二維陣列rec，並用布林值判斷測資是否重複，若是重複就跳出迴圈重新洗牌;若沒有重複就將測資記錄進rec中:

(詳細實作程式請參考附錄: Q2_InsertionSort.cpp、Q2_SelectionSort.cpp、Q2_BubbleSort、Q2_QuickSort)

```
int rec[60][5];
for(int i = 0; i < 60; i++){
      while(1){
            bool alr = 0;
            random_shuffle(array, array + 5);
            for(int j = 0; j < i; j++){
            int samenum=0;
```

```
            for(int k = 0; k < 5; k++){
                if(rec[j][k] == array[k]){
                    samenum++;
                }
            }
            if(samenum == 5){
                alr = 1;
                break;
            }
        }
        if(!alr){
            for(int j = 0; j < 5; j++){
                rec[i][j] = array[j];
            }
            break;
        }
    }
}
```

若以此研究報告的主程式來舉例，將陣列中的元素設定為$rand()$代表此函數回傳的隨機值將落在0到$RAND_MAX$之間，而電腦其實沒辦法真的隨機產生亂數，而是進行一系列複雜的運算來模擬亂數的產生。但因為程式中固定了亂數種子$srand(7)$，也就是固定了初始值，如此就能確保四種執行方式所使用的測資皆為相同，也保證了比較的公平性。

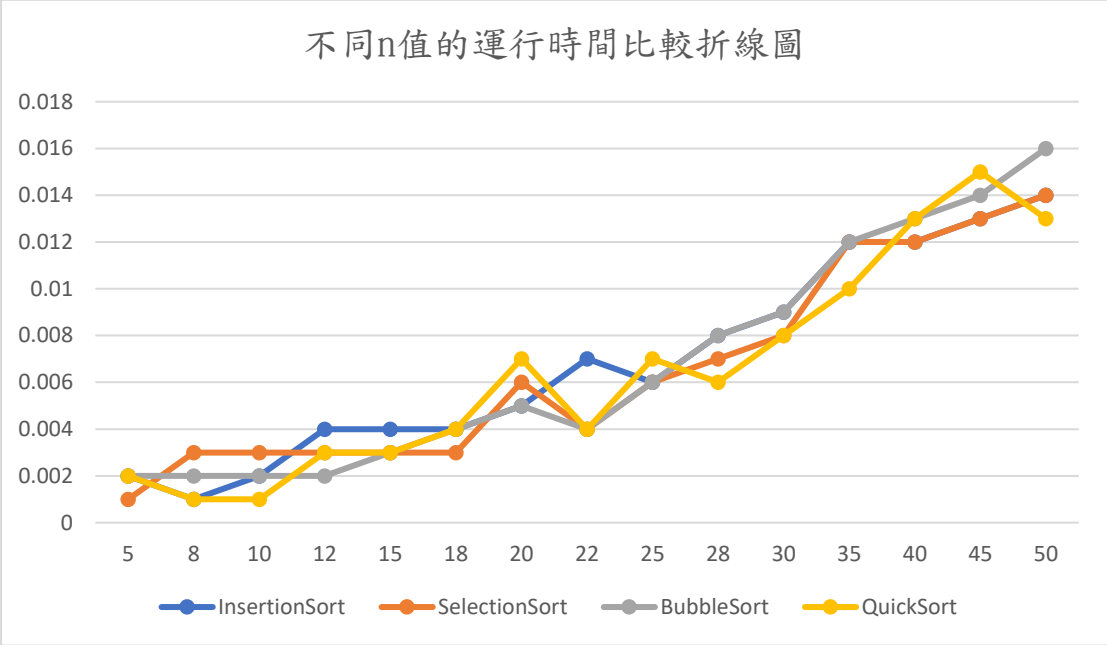
4. 已有部分研究報告顯示，大小不同的n值，上述四種方法的執行時間互有優劣。沒有那一個絕對是最好的。請依測試結果（沒有 100%的標準答案）闡述之。

宣告一個常數整數陣列事先設定好我想要的各個n值進行比較，並且同樣使用$rand$函數固定亂數種子以確保四種 sorting 的測試資料相同:
(詳細實作程式請參考附錄:Q4_InsertionSort.cpp、Q4_SelectionSort.cpp、Q4_BubbleSort.cpp、Q4_Quicksort.cpp)

```
const int num[15] = {5, 8, 10, 12, 15, 18, 20, 22, 25,
28, 30, 35, 40, 45, 50};
for(int i = 0; i < 15; i++)
```

```
{
int n = num[i];
}
```

將實作時間做成以下圖表分析可以發現，不同大小的n值在四種執行方法下真的各有優劣，造成折線崎嶇且四種 sorting 各自有執行效率最佳的時候。



5. 如何避免程式執行時發生正數+正數=負數的情況?

因為 int 的數值範圍有限，為-2,147,483,648 至 2,147,483,647，因此若是累加值大於最大整數值，就會出現 overflow 而發生正數+正數=負數的情況。若要避免這種狀況，可以宣告一個常數變數值為 INT_MAX，並將此變數定為迴圈最高運行次數，如此便不會出現累加值為負的狀況:

```
const int nMax = INT_MAX;
      for(int i = 0; i < nMax; i++)
      {
      ……
      }
```

又或者可以選擇用第二種方法:在程式一開始宣告變數型態時就直接設定為 long long int 型態而非 int 型態，如此一來累加值的範圍便可擴大為 -9,223,372,036,854,775,808 至 9,223,372,036,854,775,807，也就不會輕易地

出現 overflow 的狀況了。

6. 同一個n值，除了取測試筆數執行時間平均值外，也可以求其最大值、最小值。

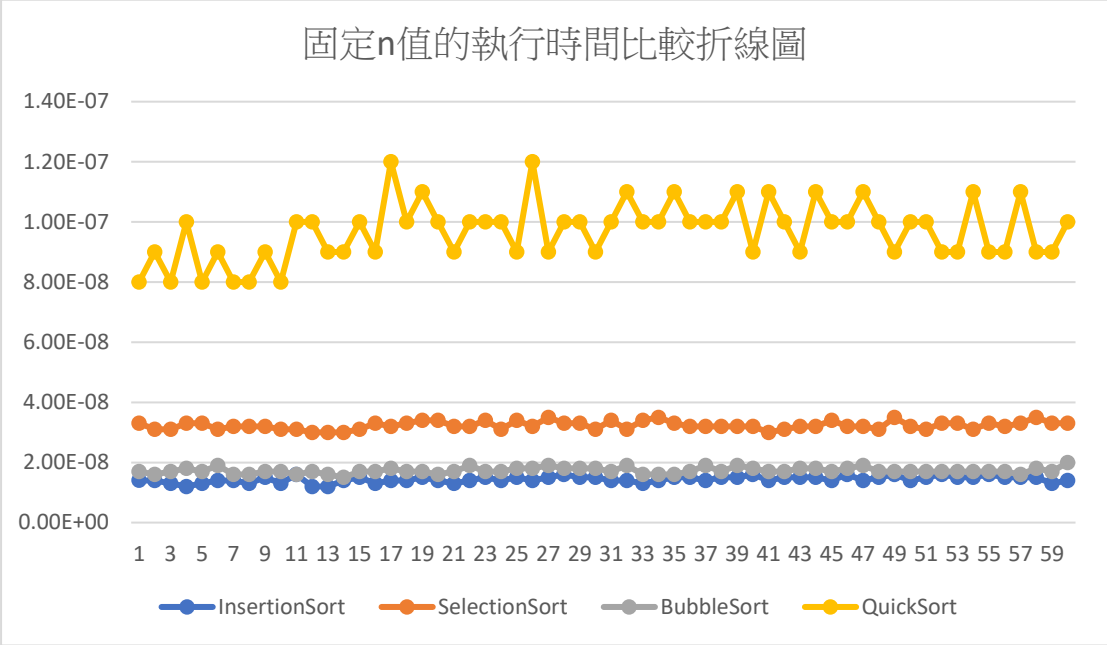
將第二題的程式碼稍微修改成，在 60 次的運算中每次皆印出計算時間，便可判斷執行時間的最大值與最小值:

```
for(int j = 0; j < 60; j++)
     {
     random_shuffle(array, array + 5);
     ……
     cout << "The sorting time: " << …… << endl;
  }
```

由於此三種撰寫方式的單次執行時間非常快速，無法有效取得，必須重複執行數次，最後在輸出時再將計算出的時間除以迴圈執行的次數。

```
for(int j = 0; j < 60; j++)
  {
     for(int p = 0; p < 1000000; p++)
     {
     ……
     }
     cout << "The sorting time: " << ((float)clicks /
     CLOCKS_PER_SEC) / 1000000 << endl;
  }
```

由程式實作的測試結果如下:



	Maximum sorting time	Minimum sorting time
Insertion sort	0.000000016s	0.000000012s
Selection sort	0.000000035s	0.00000003s
Bubble sort	0.00000002s	0.000000015s
Quick sort	0.00000012s	0.00000008s

7. 執行效率除了以執行時間表示外，亦可標定關鍵的運算執行次數。在本研究主題中，關鍵的運算可標定為兩個數比大小">"或"<"。測試程式可以統計"比較" 的執行次數。

同樣大小的n，因測試資料原始排列不同，會造成標定的關鍵運算執行次數不同，因此下列程式碼皆用相同的亂數種子以及固定的n，以公平的比較關鍵運算之執行次數

```
Int main(){
srand(7);
int a = rand(), b = rand(), c = rand(), d = rand(), e
= rand();
int array[5] = {a, b, c, d, e};
……}
```

各個版本得到的結果分別是:

實測結果	關鍵運算執行次數
Insertion sort	4 次
Selection sort	3 次
Bubble sort	4 次
Quick sort	6 次

由以上數據可以分析出，在此設定情況下，執行效率最佳的為 Selection sort，反之則為 Quick sort。

以下附上「計算各種 sorting 的關鍵運算執行次數」之函式程式碼截圖:

Insertion sort 版本:

```
int count = 0;
void InsertionSort(int *arr, int size)
{
    for(int i = 1; i < size; i++)
    {
        int tmp = arr[i];
        int j = i - 1;
        while(tmp < arr[j] && j >= 0)
        {
            count++;
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = tmp;
    }
}
```

Selection sort 版本:

```
int count = 0;
void SelectionSort(int *arr, int size)
{
    for(int i = 0; i < (size - 1); i++)
    {
        int min = i;
        for(int j = i + 1; j < size; j++)
        {
            if(arr[j] < arr[min])
            {
                count++;
                min = j;
            }
        }
        int tmp = arr[min];
        arr[min] = arr[i];
        arr[i] = tmp;
    }
}
```

Bubble sort 版本:

```
int count = 0;
void BubbleSort(int *arr, int size)
{
    for(int i = 0; i < (size - 1); i++)
    {
        for(int j = i; j < (size - i - 1); j++)
        {
            if(arr[j] > arr[j + 1])
            {
                count++;
                int tmp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = tmp;
            }
        }
    }
}
```

Quick sort 版本:

```
int count = 0;
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int Partition(int *arr, int start, int end)
{
    int pivot = arr[end];
    int i = start - 1;
    for(int j = start; j < end; j++)
    {
        if(arr[j] < pivot)
        {
            count++;
            i++;
            swap(&arr[j], &arr[i]);
        }
    }
    i++;
    swap(&arr[end], &arr[i]);
    return i;
}

void QuickSort(int *arr, int start, int end)
{
    if(start < end)
    {
        count++;
        int pivot = Partition(arr, start, end);
        QuickSort(arr, start, (pivot - 1));
        QuickSort(arr, (pivot + 1), end);
    }

}
```

8. 承 6 與 7，如果測試筆數夠多，四種方法的"比較"的執行次數最大值、平均值、最小值會與 n^2, $n\log n$有關。

以下以表格分別介紹四種方法的"比較"的執行次數最大值、平均值、最小值與 n^2, $n\log n$之關聯性。

Insertion sort:

	時間複雜度	說明
Best case	1	當資料的順序恰好為由小到大時，每回合只需比較 1 次
Worst case	n^2	當資料的順序恰好為由大到小時，第 i 回合需比 i 次
Average case	n^2	第 n 筆資料，平均比較$\frac{n}{2}$次

Selection sort:

	時間複雜度	說明
Best case	n^2	無論資料順序如何，都會執行兩個迴圈
Worst case	n^2	
Average case	n^2	

Bubble sort:

	時間複雜度	說明
Best case	n	當資料的順序恰好為由小到大時 第一次執行後，未進行任何 swap ⇒ 提前結束
Worst case	n^2	當資料的順序恰好為由大到小時 每回合分別執行：$n-1$、$n-2$、...、1次 $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$
Average case	n^2	第 n 筆資料，平均比較$\frac{(n-1)}{2}$次

Quick sort:

	時間複雜度	說明
Best case	$n\,log\,n$	第一個基準值的位置剛好是中位數，將資料均分成二等份
Worst case	n^2	當資料的順序恰好為由大到小或由小到大時有分割跟沒分割一樣
Average case	$n\,log\,n$	