

資料結構

HW4

系級：電機系大三

姓名：林凡皓

學號：B103012002

一、 Bubble sort in both directions

利用 while 迴圈來做排序，while 迴圈會一直執行直到遍歷整個數列後都沒有交換產生。為了要做雙向的遍歷數列，我利用 start 和 end 參數分別代表正向遍歷與反向遍歷的起始位置，最初 start 指向數列的 index 0，而 end 指向數列的倒數第二個 index。每遍歷完一次數列，start 指向右邊的位置，end 指向左邊的位置。

此外，要根據 descending 的值決定遞增或是遞減排序。如果是遞增，在正向遍歷的時候要比較下一個元素與當前元素，如果下一個元素較小就要做交換，反向遍歷則是比較當前元素與左邊的元素，如果左邊的元素較大，就交換。如果是遞減排序，則判斷是否交換的條件要相反。

測試函數功能的部分，對數列[20, 30, 40, 90, 50, 60, 70, 80, 100, 110]做排序，並將結果與正確結果比較，測試結果如下

```
bubble_sort_bidirection ascending correct !
bubble_sort_bidirection descending correct !
```

不論是遞增或是遞減都能夠正確執行。

二、 Stable selection sort

先前 unstable selection sort 之所以會 unstable 是因為演算法利用交換來做排序，為了避免掉交換，我改用插入的概念，即將元素直接插入至對應位置，而非一步一步交換，如此一來演算法就會變成 stable。

為了要驗證修改過後以及修改前的演算法差別，我將教授提供的 select_sort 改為可以選擇要對 tuple 中哪一個元素做排序。執行結果如下

```
Original list : [(5, 'A'), (3, 'A'), (2, 'A'), (4, 'A'), (6, 'A'), (1, 'A'), (2, 'B')]

Result of select_sort_stable
key = 0 : [(1, 'A'), (2, 'A'), (2, 'B'), (3, 'A'), (4, 'A'), (5, 'A'), (6, 'A')]
key = 1 : [(5, 'A'), (3, 'A'), (2, 'A'), (4, 'A'), (6, 'A'), (1, 'A'), (2, 'B')]

Result of select_sort
key = 0 : [(1, 'A'), (2, 'B'), (2, 'A'), (3, 'A'), (4, 'A'), (5, 'A'), (6, 'A')]
key = 1 : [(5, 'A'), (3, 'A'), (2, 'A'), (4, 'A'), (6, 'A'), (1, 'A'), (2, 'B')]
```

由結果可以看到，對於 key = 0 的情況來說，原先的 list 為(2, 'A')在(2, 'b')前面，經果 select_sort_stable 的排序後，此順序保持不變，但是未經修改 select_sort 會將順序顛倒。對於 key = 1 的情況來說，select_sort_stable 依然會保持相同元素原先的相對位置。

三、 Implement median of three for quick sort

此演算法與原先的 quick sort 差別主要是在 pivot 的選擇。為了要取的第一個、最後一個和中間元素的中位數，我自行寫了一個 function 來協助完成這項任務。

將利用 median of three 取的 pivot 的演算法與先前的演算法在很長的數列做比較，結果如下表格

quicksort_medain	<pre>Running quick_sort_median on array of size 50000... Completed quick_sort_median.</pre>
quick_sort	<pre>RecursionError: maximum recursion depth exceeded PS C:\Users\USER\Desktop\HW4></pre>

由結果可以看到，如果沒有使用 median of three，會發生 maximum recursion depth exceeded 的問題，但是修改過後就可以順利執行。

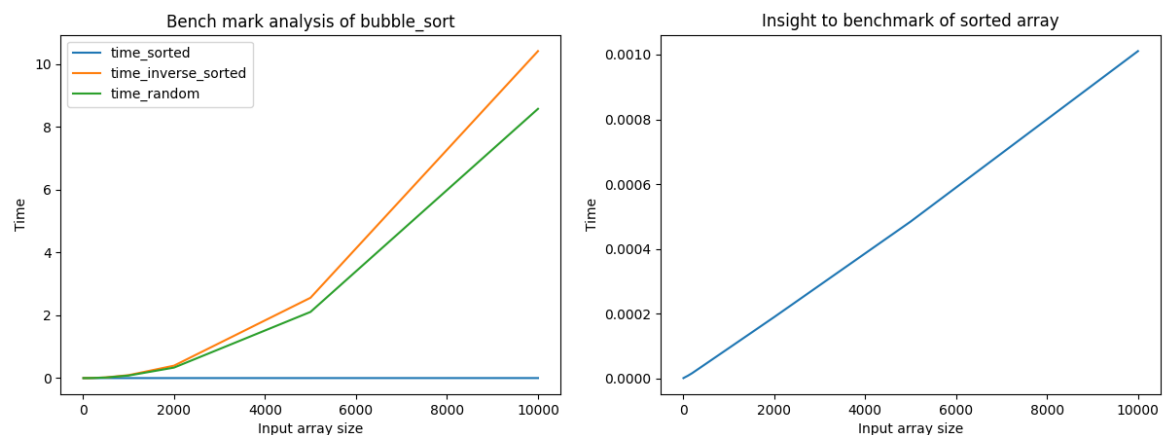
四、 Benchmark analysis

- Bubble sort

理論上，bubble sort 對於一個隨機的數列做排序的時間複雜度為

$O(n^2)$ 。Bubble sort 最佳情況為對一個已經排列好的數列做排序，其時間複雜度為 $O(n)$ ，bubble sort 最差情況為對一個反著排序的數列做排序，其時間複雜度為 $O(n^2)$ 。

Benchmark analysis 的部分，主要是對 9 種不同大小(10, 50, 100, 200, 500, 1000, 2000, 5000, 10000) 的數列做排序，同時也比較不同情形的數列(已排列、反著排列、隨機)做排列時間上的比較。將結果視覺化後如下



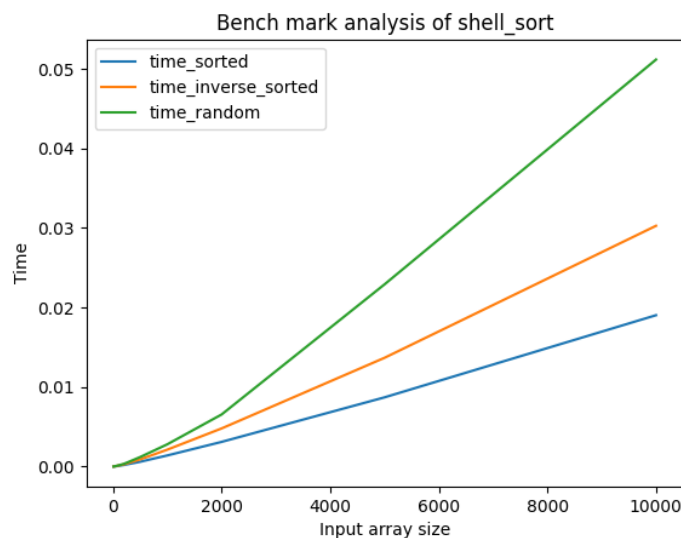
由左圖可以看出，對於已排列好的數列來說，運算時間非常短，而對

於隨機數列以及反著排列的數列來說，時間是隨著數列大小增加成平方增加，且反著排列的數列所花費的時間比隨機數列還要高。

做途中因為 scale 的關係，無法看出對於已排列好的數列做排列的時間複雜度，因此我特別加了右圖，方便觀察結果。從右圖中，可以清楚看到對於已排列好的數列，運算時間隨著數列大小成線性增加。

- Shell sort

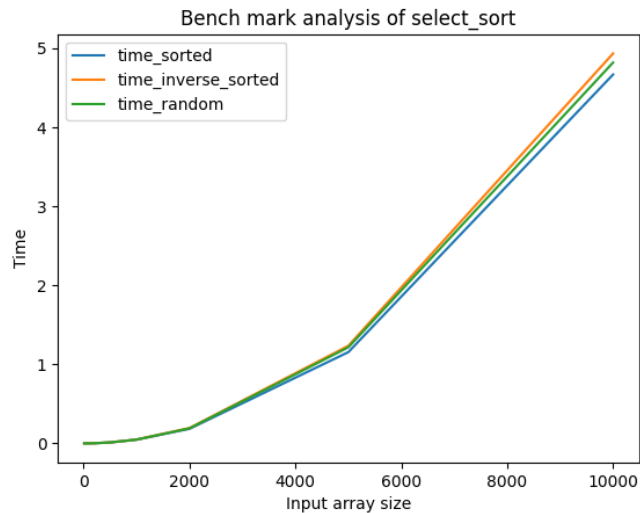
Shell sort 的時間複雜度取決於 gap sequence 如何安排，一般情況會介於 $O(n^2) \sim O(n)$ 。將 benchmark analysis 結果視覺化後如下



由結果可以看出，對於不同情況的數列，在 shell sort 中的排序時間差距比較小，運算時間隨數列大小成指數增加，但是指數次方比 bubble sort 還要小。

- Selection sort

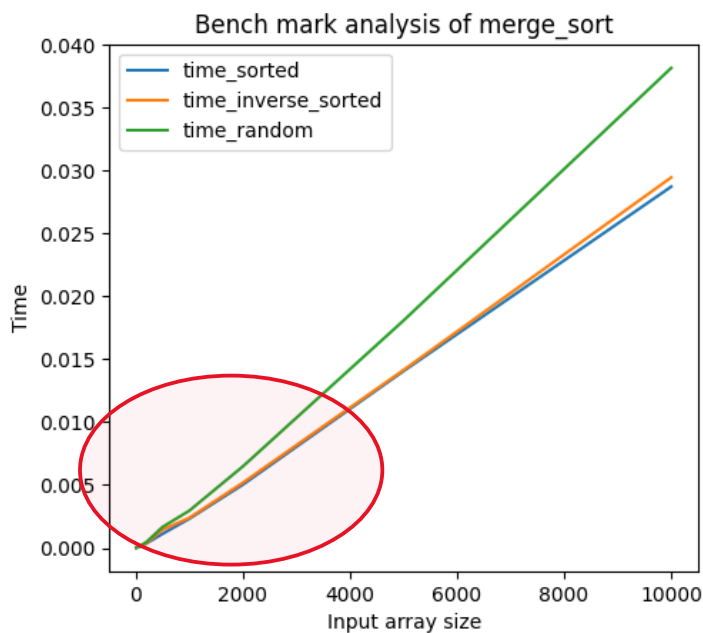
理論上，selection sort 對於不同情況的數列會有差不多的表現，時間複雜度為 $O(n^2)$ 。將 benchmark analysis 結果視覺化後如下



由結果可以明顯看出，對於三種情況的數列做排序所花費的時間差不多，運算時間隨數列大小增加成平方增加。

- Merge sort

理論上，merge sort 對於不同情況的數列之時間複雜度皆為 $O(n \log n)$ 。將 benchmark analysis 的結果視覺化後下



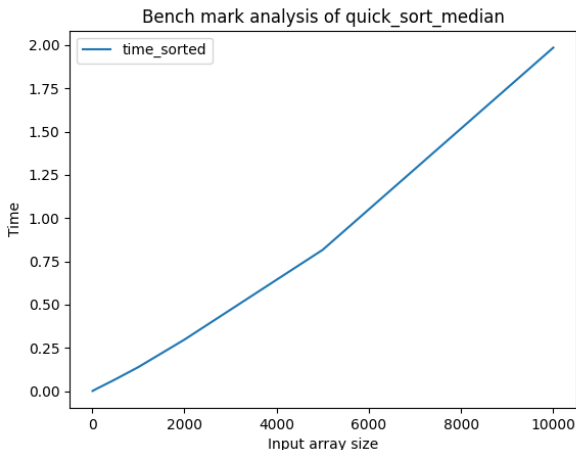
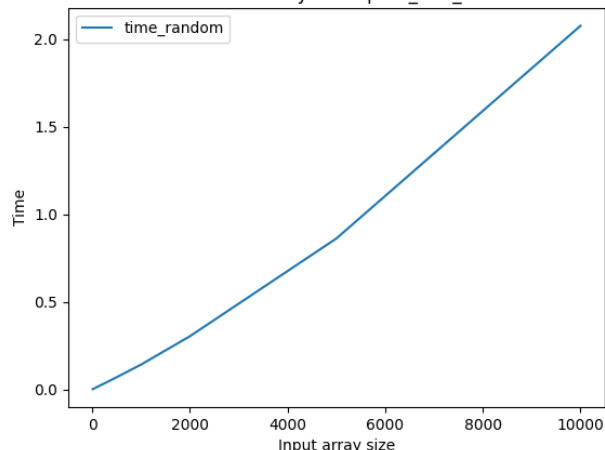
在紅色框起來的部分中，可以隱約看出運算時間隨著數列大小增加約比線性增加再快一點，符合 $O(n \log n)$ 。

- Quick sort

由於未經優化過的 quick sort 會有 stack overflow 的問題，因此採用作業中實作出來的 quick_sort_median 來作 benchmark analysis。

理論上，quick_sort_median 對於不同情況的數列之時間複雜度皆為 $O(n \log n)$ 。由於同時跑三種不同情況的數列的分析會有 stack overflow

的問題，因此採用分開分析，將結果視覺化後如下

Sorting on sorted array	 <p>Bench mark analysis of quick_sort_median</p> <p>time_sorted</p> <p>Time</p> <p>Input array size</p>
Sorting on inversed sorted array	<pre>File "c:\Users\User\Desktop\大三下\資料結構\HW\HW4\ds_sorting.py", line 195, in quicksort_median quicksort_median(array, leftindex, newpivotindex) [Previous line repeated 987 more times] File "c:\Users\User\Desktop\大三下\資料結構\HW\HW4\ds_sorting.py", line 194, in quicksort_median newpivotindex = partition_median(array, leftindex, rightindex) ~~~~~ File "c:\Users\User\Desktop\大三下\資料結構\HW\HW4\ds_sorting.py", line 155, in partition_median pivot = median(left, right, middle) ~~~~~ RecursionError: maximum recursion depth exceeded</pre>
Sorting on random array	 <p>Bench mark analysis of quick_sort_median</p> <p>time_random</p> <p>Time</p> <p>Input array size</p>

由結果可以看到對於 **inversed sorted array** 來說，即便修改成 **median three quick sort** 之後還是會有 **stack overflow** 的問題，主要是因為 **left mark** 每動一次就會發現到下一個元素大於 **pivot value**，因此迴圈會執行非常多次，導致 **stack overflow** 的問題。

不看這種特殊情況下，可以看到**運算時間隨著數列大小增加約比線性增加再快一點**，符合 $O(n \log n)$ 。

五、心得

這次作業主要是修改上課所教的排序演算法以及對這些演算法做比較與 **benchmark** 分析。在修改這些演算法之前，我們需要對這些演算法有足夠

程度的了解才可以依照我們的需求作修改，因此我花了許多時間在理解這些上課所教的演算法，也透過這個過程對這些演算法有了更深的了解與印象。此外，透過 benchmark 分析以及視覺化，對於上課時利用數學推導出的時間複雜度分析也有了更深的印象，同時也對 benchmark 分析的方法、code 以及一些視覺化工具的使用更加熟練。