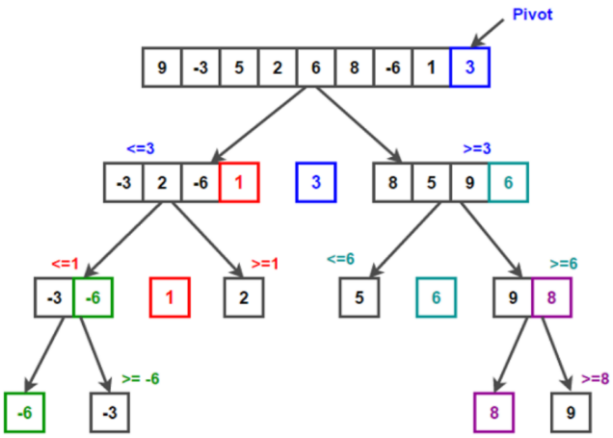
演算法導論 HW1

110511254 徐煜絨

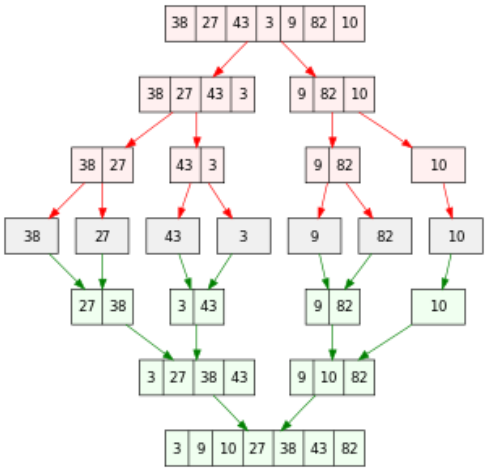
一、程式架構

這次作業要實現 randomized quicksort，關於 quicksort 的架構和執行方式如下圖：



（圖源：https://www.techiedelight.com/quicksort/）

而下面這張則是 merge sort：



（圖源：https://en.wikipedia.org/wiki/Merge_sort）

Quick sort 和 Merge sort 最大的不同在於其分堆方式，merge sort 只是把 array 以 recursive 持續分成兩等分，最後再依序比較兩份中的值並組合。Quick sort 則是會在 array 中先選定 pivot，把比 pivot 大和比 pivot 小的分成左右兩堆，接著在左右兩堆中再分別找新的 pivot，持續分堆。如此，只需要在同一個 array 上處理資料，可以節省記憶體空間，也不用像 merge sort 一樣，最後合併時尋找應該放入哪個元素。

二、程式碼解釋

```
void RM_Quicksort(int *array1, int p, int r){
    if (p<r){
        int q = RM_Partition(array1, p, r);
        RM_Quicksort(array1, p, q-1);
        RM_Quicksort(array1, q+1, r);
    }
}
```

在 RM_Quicksort 中，要先判斷 p 是否小於 r。接著，q 代表被設為 pivot 的那個位置，所以下面兩行 RM_Quicksort 的開頭會是 p，並且結束在 q-1，另一行則是由 q+1 開始到 r。

These values are <= pivot			pivot	These values are > pivot			
p	…	q-1	q	q+1	…	…	r
進入下一個 RM_Quicksort			—	進入下一個 RM_Quicksort			

```
int RM_Partition(int *array1, int p, int r){
    int i = rand()%(r-p+1)+p;  // p~r
    swap(array1[r], array1[i]);
    return Partition(array1, p, r);
}
```

在 RM_Partition 中，會從 p 到 r 間找一數字 i，並把 array1[r]和 array1[i]對調。經過這個步驟，pivot 就不一定是 array1[r]的值，而是 random 的，如次有機會縮短 quick sort 的執行時間。

```
int Partition(int *array1, int p, int r){
    int x = array1[r];
    int i = p-1;
    for(int j=p; j<r; j++){
        if(array1[j] <= x){
            i++;
            swap(array1[i], array1[j]);
        }
    }
    swap(array1[i+1], array1[r]);
    return i+1;
}
```

在 Partition 中，pivot 會落在 i+1 這個位子，而 p 到 i 位置上的 value 會<= pivot，i+2 到 r 位置上的 value 則>pivot。最後回傳 pivot 的位置，即 i+1。

三、時間複雜度

1. Quick sort

在 Quick sort 中因為涉及 pivot 的選擇，所以時間複雜度的計算會較複雜。下面有兩種 case：

第一種，假設 x 介在Z_i和Z_j間，即$Z_i < x < Z_j$，則Z_i和Z_j兩者，在之後的 partition 中都不會互相被比較。第二種，當Z_i或Z_j被選為 pivot，則他會被拿來和Z_{ij}中的其他 elements 做比較。

由上可發現，Z_i和Z_j兩者能互相比較的機率相當於Z_i或Z_j被選為 pivot 的機率-t，又等於Z_i被選為 pivot 的機率加上Z_j被選為 pivot 的機率（以下記作 Pr）。

$$Pr = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$\text{期望值 } E[x] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k} = \sum_{i=1}^{n-1} O(logn) = O(nlogn)$$

所以 Quick sort 的 time complexity 的平均值是 O(nlogn)。在 worst case 時，Partition 這個 function 裡面的 swap 會每次都執行，發生在 input 是排好的陣列且選擇最後一個 index 做為 pivot，此時的時間複雜度是 O(n^2)。

2. Merge sort

Merge sort 中有遞迴和排列，平均和最糟的時間複雜度都是 O(nlogn)。

$T(n) = 2T(n/2) + D(n) + C(n)$ (divide) (combine)

$= 2T(n/2) + \Theta(1) + \Theta(n)$ → cn

$= cn\log_2 n + cn$

$\therefore T(n) = \Theta(n\log_2 n)$

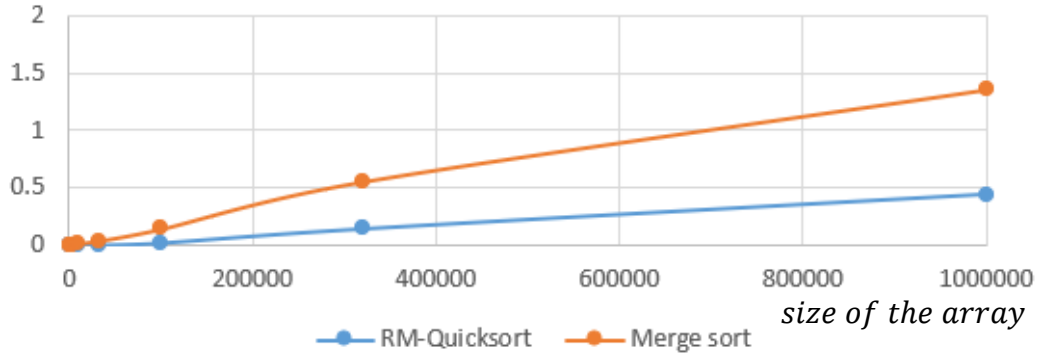
四、實際測試

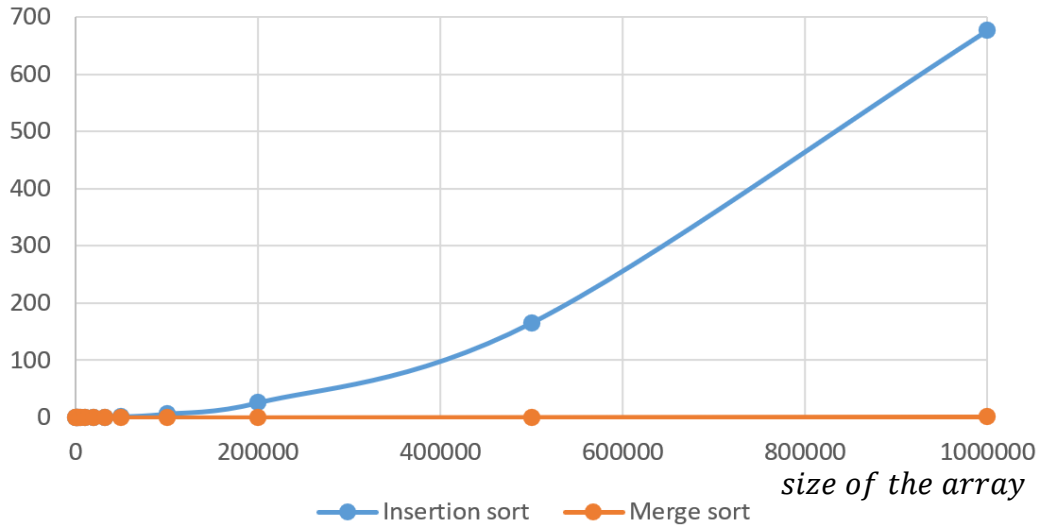
```
Please enter the size of the array: 10000
randomized quicksort time: 0.003
```

<- 輸出結果截圖

Size of the array	RM_Quicksort (sec.)		Merge sort (sec.)	
	srand(5)	srand(30)	srand(5)	srand(30)
100	0	0	0	0
500	0	0	0.001	0.001

1000	0	0	0.001	0.001
3200	0.001	0	0.004	0.006
10000	0.002	0.003	0.022	0.017
32000	0.006	0.009	0.04	0.043
100000	0.03	0.022	0.148	0.141
320000	0.2	0.096	0.611	0.501
1000000	0.435	0.455	1.365	1.348



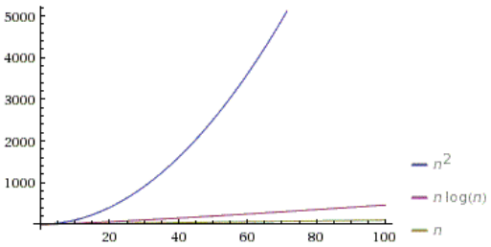


上表橫軸為陣列大小，縱軸為執行時間。程式中 srand 的 seed 原本是 time(NULL)，但為了測試執行時間，我把 seed 設成 5 和 30 並分別測試，圖表中的結果是由兩者做平均所得。Seed 設為定值的原因是為了在測試時，陣列內容能有一定相似性，隨著陣列大小增加，陣列會在保有原先內容的情況下被加上其他數值。

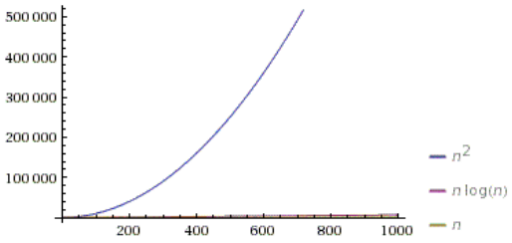
觀察上圖，上升程度雖然看似 n，但因為$\log n < n^{\varepsilon}$ for any $0 < \varepsilon < 1$，在 size 不夠還有 rand()的操作下，可能造成 n 和 nlogn 看起來相似，而經過理論推導，可以知道 RM_Quicksort（正常狀況下）和 Merge sort 都是 nlogn 的演算法。下圖則是 Insertion sort 和 Merge sort 的比較，可以清楚看見 n^2 和 nlogn 的差異程度。

下面兩張圖是 nlogn、n^2 和 n 的圖，左邊為 nϵ[0:100]，右邊則是 nϵ[0:1000]。

nϵ[0:100]



nϵ[0:1000]



圖源：

https://stackoverflow.com/questions/23329234/which-is-better-on-log-n-or-on2