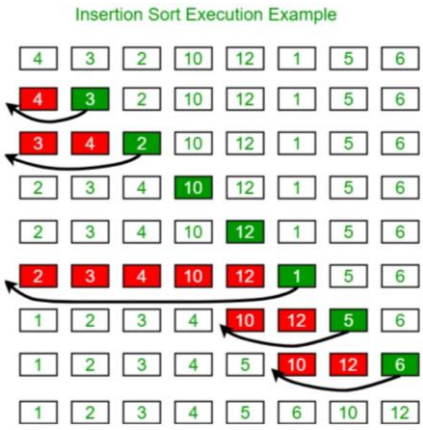
演算法導論 HW1

110511254 徐煜絨

一、程式架構

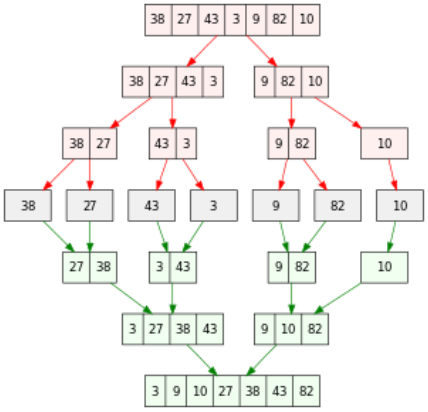
除了主程式，另外還有四個函式，包含在陣列中生成隨機變數、insertion sort, merge sort 和 merge。

首先是 insertion sort，其排序法為預設第 i 位前已經排好，再將第 i+1 位的值與前面（[0]~[i]做比較）。若希望最後結果是由小到大，第 i+1 位的值會被令為 key，當 key 和第 i 位比較而 key<a[i]時，那 a[i]的值就會被寫入 a[i+1]，而 key 會繼續和 a[i-1]比大小。當 key>=a[n]，key 會被寫到 a[n+1]，此時 i=i+1，新的 a[i+1]被令成 key…，持續重複以上動作直到 i=size-1（當陣列大小=size，最後一個位子的編號會是 size-1）。



（圖源：https://reurl.cc/qrAv9N）

Merge sort 則是把一開始的陣列多次分成兩部分，最後再組合起來。從下圖可以看到拆解的過程是 recursive 的，拆分的程度取決於程式碼，下圖的 base case 和我的程式碼都是當拆解到剩一個元素再開始 merge。



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

最後是主程式，我用 srand(time(NULL))以及 gen 函式中的 rand()生成陣列內的變數。在時間計算上則是在呼叫 insertion sort 和 merge sort 函式的前後，用 clock()/CLOCKS_PER_SEC 去取當下的時間，再把兩者相減而得。

二、Pseudo code 和程式碼比較與解釋

1. Insertion sort

Pseudo code:

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Code in practice:

```
void insertion_sort(int *a, int size){
    for(int i=1; i<size; i++){
        int key = a[i];
        int j = i-1;
        while(j>=0 && a[j]>key){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = key;
    }
}
```

兩者的差別在於上面是將 a[1]當作 array 的第一位，而我是把 a[0]當作第一位，所以 pseudo code 從 j=2 to A.length，我則是寫 i=1; i<size; i++。另外在 while-loop 的條件式中，跟前面的原因相同，所以當 j 為 0 時，也要執行 while 迴圈。

2. Merge sort

Pseudo code:

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

Code in practice:

```
void merge_sort(int *a, int p, int r){
    int q;
    if (p>=r) return;
    q = (p+r)/2;
    merge_sort(a, p, q);
    merge_sort(a, q+1, r);
    merge(a, p, q, r);
}
```

相比於 pseudo code，實際上會在有遞迴的程式中加入 base case，以 merge sort 為例，加上 if (p>=r) return，可以讓拆分到剩下一個 element 時，因為 p==r，所以結束遞迴並開始 merge。

在 merge 時會因為陣列設定的方式不同，導致執行可能出現 segmentation fault，如下圖。一開始是使用第一行做設定，然而當陣列大小達到 1000000，在執行時就會跳出警告，原先以為是電腦容量不夠，在查完資料後發現若是以 Array_name[size]產生新陣列，因為是靜態陣列，其大小會有一定限制，所以最後改用動態陣列，並順利解決問題。

```
// int L[n1], R[n2];
int * L = new int[n1];
int * R = new int[n2];
```

老師上課提供 merge 的 pseudo code，但因為在寫程式時不確定要怎麼讓 L[n1], R[n2] -> ∞，所以我這裡用另外一種方法，讓 L 或 R 其中一方被完全排進 A 後，另外一方剩下的值能全部進到 A。

```
while(i<n1 && j<n2){
    if(L[i] <= R[j]){
        a[k] = L[i];
        i++;
    }
    else{
        a[k] = R[j];
        j++;
    }
    k++;
}
while(i<n1){
    a[k] = L[i];
    i++;
    k++;
}
while(j<n2){
    a[k] = R[j];
    j++;
    k++;
}
```

當 L 和 R 都還有值

R 被取完，剩下 L 有值

L 被取完，剩下 R 有值

Merge

Pseudo

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

In practice

```
void merge(int *a, int p, int q, int r){
    int i, j, k;
    int n1 = q-p+1;
    int n2 = r-q;
    // int L[n1], R[n2];
    int * L = new int[n1];
    int * R = new int[n2];
    for(i=0; i<n1; i++){
        L[i] = a[p+i];
    }
    for(j=0; j<n2; j++){
        R[j] = a[q+1+j];
    }
    i = 0;
    j = 0;
    k = p;
    while(i<n1 && j<n2){
        if(L[i] <= R[j]){
            a[k] = L[i];
            i++;
        }
        else{
            a[k] = R[j];
            j++;
        }
        k++;
    }
    while(i<n1){
        a[k] = L[i];
        i++;
        k++;
    }
    while(j<n2){
        a[k] = R[j];
        j++;
        k++;
    }
}
```

三、時間複雜度

1. Insertion sort

因為有一層 while-loop 在 for-loop 中，所以其 worst case 的時間複雜度 $T(n) = O(n^2)$。

2. Merge sort

Merge sort 中有遞迴和排列，因此計算會較為複雜。

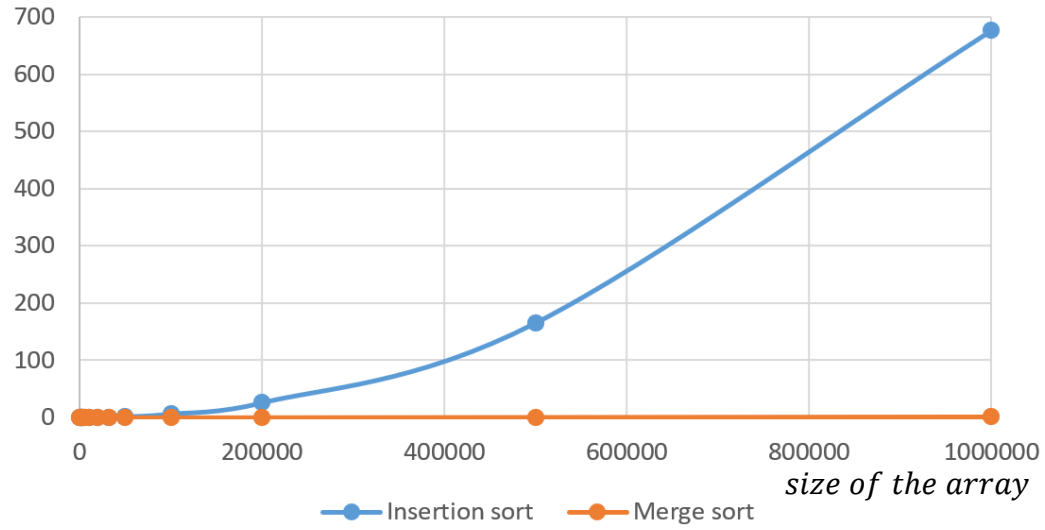
$T(n) = 2T(n/2) + D(n) + C(n)$ (divide) (combine)
$= 2T(n/2) + \theta(1) + \theta(n)$ → cn
$= cn\log_2 n + cn$
$\therefore T(n) = \theta(n\log_2 n)$

四、實際測試

```
Please enter the size of the array: 5000
insertion time: 0.018
merge time: 0.006
```

<- 輸出結果截圖

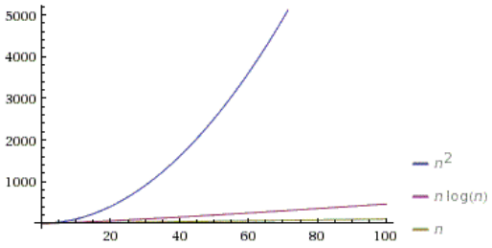
Size of the array	Insertion sort (sec.)		Merge sort (sec.)	
	srand(5)	srand(30)	srand(5)	srand(30)
100	0	0	0	0
500	0	0	0.001	0.001
1000	0.001	0.001	0.001	0.001
2000	0.003	0.004	0.002	0.003
3200	0.008	0.007	0.004	0.006
5000	0.018	0.017	0.006	0.007
10000	0.093	0.104	0.022	0.017
20000	0.294	0.296	0.024	0.025
32000	0.716	0.691	0.04	0.043
50000	1.672	1.651	0.065	0.069
100000	6.777	6.676	0.148	0.141
200000	26.351	26.152	0.278	0.275
500000	167.161	164.394	0.664	0.663
1000000	677.967	675.519	1.365	1.348



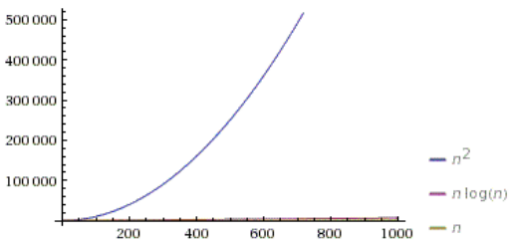
上表橫軸為陣列大小，縱軸為執行時間。程式中 srand 的 seed 原本是 time(NULL)，但為了測試執行時間，我把 seed 設成 5 和 30 並分別測試，圖表中的結果是由兩者做平均所得。Seed 設為定值的原因是為了在測試時，陣列內容能有一定相似性，隨著陣列大小增加，陣列會在保有原先內容的情況下被加上其他數值。

做成圖後，可以發現其趨勢與上面推導的 Θ 結果相似，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