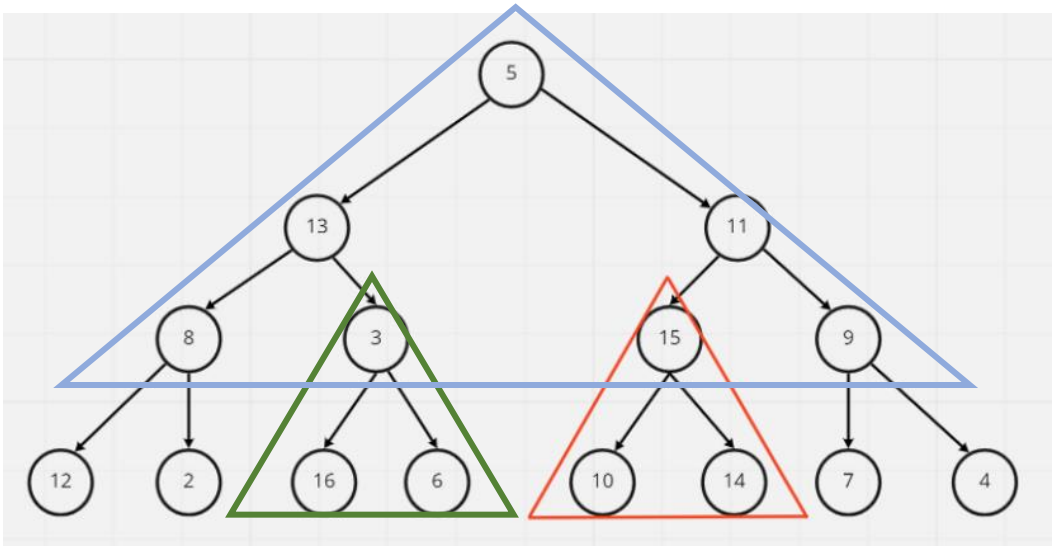
演算法導論 HW3

110511254 徐煜絨

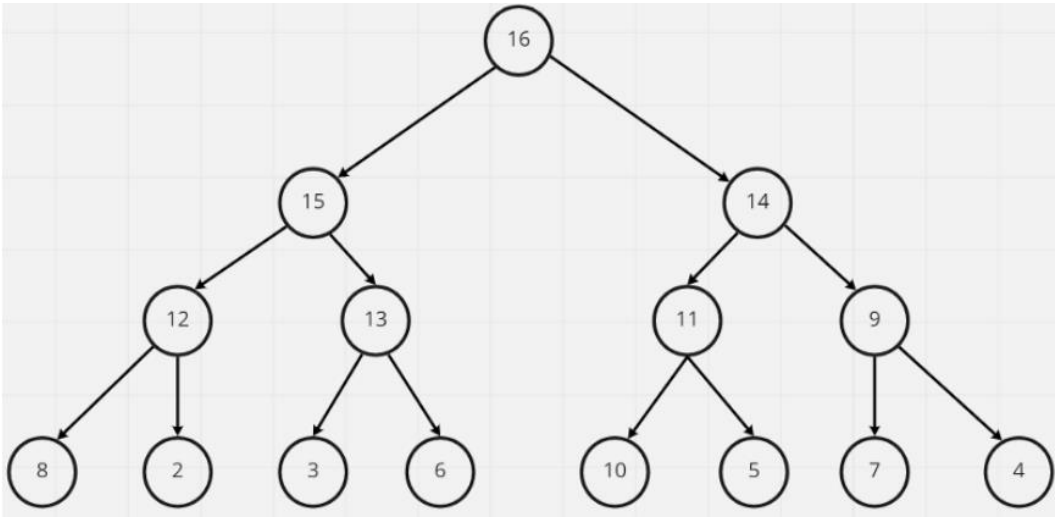
一、程式架構

在 heap sort 這個演算法中，要先讓任一三角形的上方頂點的數字，是整個三角形中最大的數字。把最大的數移到頂點的過程稱為 max-heapify。Build-max-heap 則是從最下層三角形開始做 max-heapify，並遞回到最上層。下圖中，橘色三角形已經完成 max-heapify，但藍色和綠色的三角形，其上方頂點都不是三角形中的最大值。



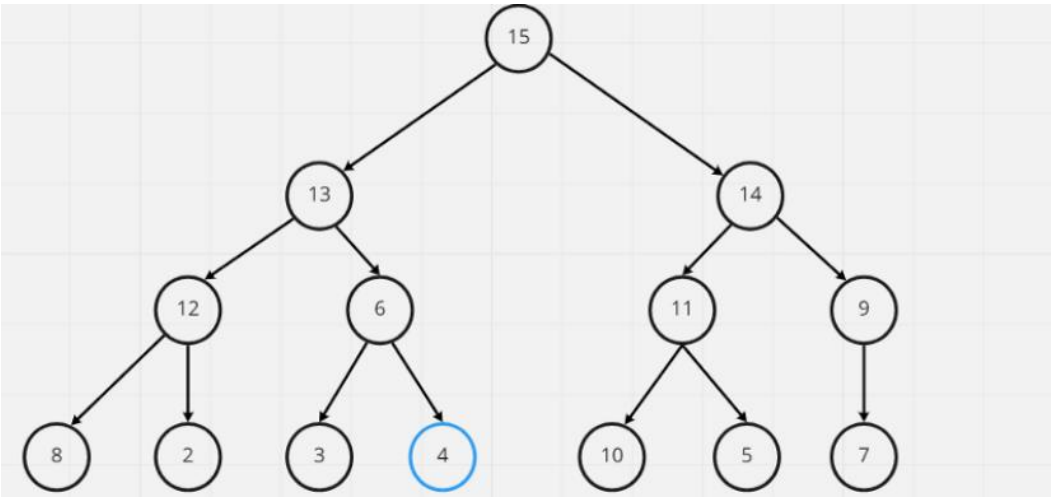
（圖片參考：https://ithelp.ithome.com.tw/articles/10266206）

下面這張圖中，任一三角形都是 max-heapify 完成的 binary tree，表示完成第一次 build-max-heap。



（圖源：https://ithelp.ithome.com.tw/articles/10266206）

建好 binary tree 後，把最上面的數字取出，並把最後一排中最右邊的數字放到頂端，再重複以上動作。以上圖為例，16 取出後，把 4 放到最頂端，在重複 max-heapify 後，最頂端會變成 15（整個 tree 中最大的 element）。此時再把 15 取出，並把 7 放到頂端，重複直到所有數字都被取出。



（圖源：https://ithelp.ithome.com.tw/articles/10266206）

二、Code 解釋

Code	Explanation

`void heap_sort(int *array1, int size){`
`  build_max_heap(array1, size);`
`  for(int i=size-1; i>0; i--){`
`  int temp = array1[0];`
`  array1[0] = array1[i];`
`  array1[i] = temp;`
`  size--;`
`  max_heapify(array1, 0, size);`
`  }`
`}`
	一開始先用”build_max_heap”建好 tree，確保任一三角形中，其上方頂點的值會最大。
	以 10 個 nodes 為例，for 迴圈在跑第一次時，array1[0]中的數字會是最大的，i=9，所以 array1[0]會和 array1[10]交換，即最頂端值和最下牌最右邊交換。之後再進到 max-heapify 做出正確的 binary tree。
	第二次 for 迴圈時，array1[0]會和 array1[8]交換，由此可以發現最大值會在 array 最右側。

`void build_max_heap(int *array1, int size){`
`  for(int i = (size/2)-1; i>=0; i--){`
`  max_heapify(array1, i, size);`
`  }`
`}`
	第一次建 tree 會比較麻煩，要先從最右下角的三角形開始建，並逐層往上，所以要跑 for 迴圈。

```c
void max_heapify(int *array1, int i, int size){
    int l = i*2 + 1;
    int r = i*2 + 2;
    int largest = i;
    if(l < size){
        if(array1[l] > array1[i]){
            largest = l;
        }
    }
    if(r < size){
        if(array1[r] > array1[largest]){
            largest = r;
        }
    }
    if(largest != i){
        int temp = array1[largest];
        array1[largest] = array1[i];
        array1[i] = temp;
        max_heapify(array1, largest, size);
    }
}
```

每個小三角形逐一檢查，確保最大值能在三角形的上頂點，如果一開始上頂點就是最大值，那就停止交換；如果不是，則把最大值換上去（假設原本在[i]），並把原本在上頂點的移到[i]，再進入下一個 max-heapify，確保以 i 為上頂點的小三角形中，在[i]的值會是最大的。

三、時間複雜度

時間複雜度：

在 build-max-heap()中，必須從第(size/2 - 1)項（array[size/2 - 1]）開始跑，且要跑到第 0 項，所以其時間複雜度為 O(n)。

Max-heapify()則是任一個值可能會往下傳遞，最糟的情況是移動 $\log_2 n$ 層，所以時間複雜度為 O(log n)。

而 heap_sort()中，資料會逐項被放到陣列裡，所以時間複雜度是 O(n)。

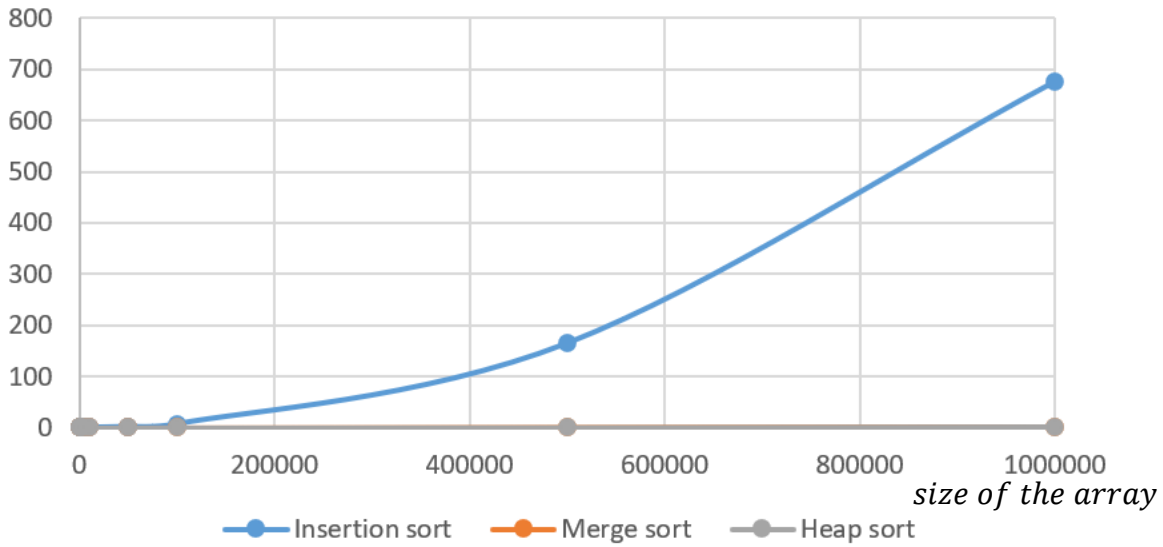
因為第一和第三個函式都會呼叫第二個函式，可以知道整個 heap sort 的時間複雜度為 O(nlog n)。

實際測試：

```
Please enter the size of the array: 5000
heap time: 0.001
```
<- 輸出結果截圖

Size of the array	srand(30)		
	Insertion sort (sec.)	Merge sort (sec.)	Heap sort (sec.)
100	0	0	0
500	0	0.001	0
1000	0.001	0.001	0
5000	0.017	0.007	0.001
10000	0.104	0.017	0.002
50000	1.651	0.069	0.012

100000	6.676	0.141	0.027
500000	164.394	0.663	0.128
1000000	675.519	1.348	0.356

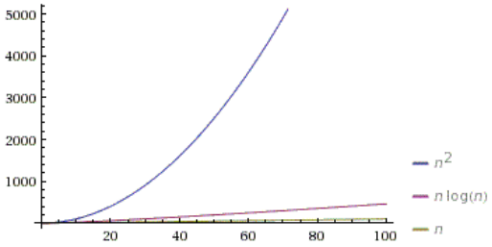


註：

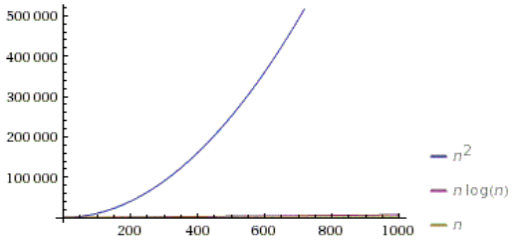
1. Insertion sort: $O(n^2)$
2. Merge sort: O(nlog n)
3. Heap sort: O(nlog n)

做成圖後，可以發現其趨勢與上面推導的時間複雜度結果相似，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）