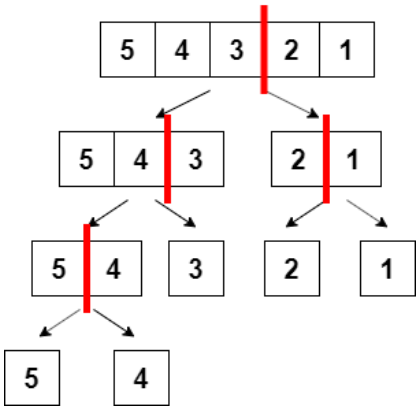
Heap Sort / Merge Sort 之比較

一、時間複雜度比較

演算法	時間複雜度		
	最佳	平均	最壞
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))

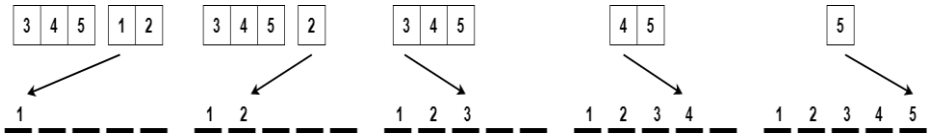
1. Merge Sort

A. 分割:

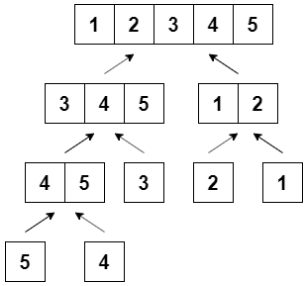


從上面的圖就可以觀察到，把一個長度為 n 的陣列做切割，切到每個小陣列都只有 1 個數字時，需要 n-1 個步驟（切 n-1 刀）。

B. 合併:



在上圖的每次排序中，兩個排序好的小陣列合併成一個排序好的大陣列時，如果兩個小陣列加起來有 n 個數字，總共就需要 n 個步驟。其中不論每個小陣列排序為何，都需走訪並排序。



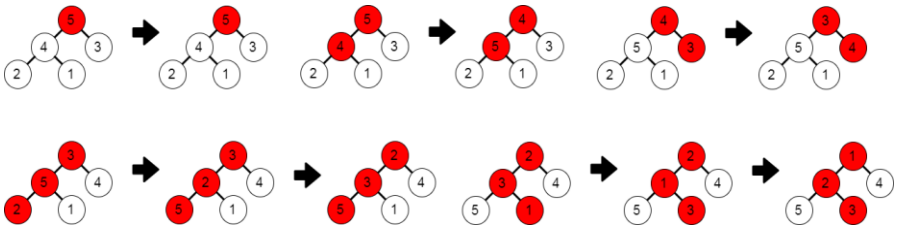
至於要經過幾回合的合併，可以想像有 1 個長度為 n 的大陣列，該陣列是經過幾次的分割才能變成 n 個小陣列，而所經過的分割次數即為所需合併的次數。以數學式表示，令 k 為所需合併的次數，n 為大陣列的長度， $1 \times 2^k = n$ ，$log_2 2^k = log_2 n$ ，$k = log_2 n$，因此所需合併的次數為 $log_2 n$。考慮每次合併需要 n 個步驟，並且須重複該步驟 $log_2 n$ 次，因此所有合併的流程需要 $nlog_2 n$ 次。

C. 分割 + 合併:

綜上所述，Merge Sort 不論何種情形，分割的流程 n-1 個步驟，再加上合併的流程 $nlog_2 n$ 個步驟，因此 Merge Sort 所有的流程需 (n-1) + $nlog_2 n$ 個步驟，其時間複雜度為 O($nlog_2 n$)。

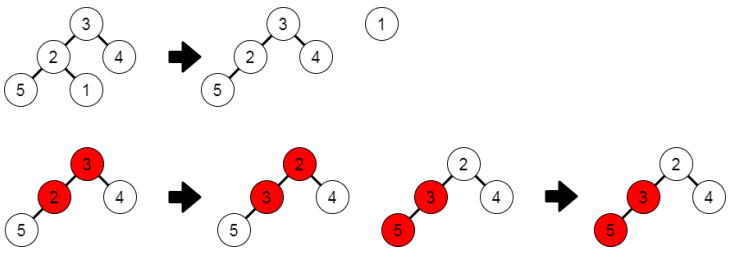
2. Heap Sort

A. 建立 Min Heap Tree



每個陣列中的數值都需實際走訪過一次，並與其父節點比較大小，如果陣列的長度為 n，那就需重複 n 次的走訪跟比較。

B. 執行 Delete Min



每一次 Delete Min 後 heap tree 的型態都會因被破壞而需重建，而重建時只需把位於根節點的數值向下依序與較小的子節點比較，因為每次只需跟最小的子節點比較，所以比較的次數即為該樹狀結構的分割次數，而樹狀結構分割的次數計算與在 Merge Sort 中合併的次數計算相同，因此每一次 Delete Min 所需的步驟為 $log_2 n$ 次。另外，Heap Sort 會在除了根節點外所有節點都被剔除後而停止，因此會經過 n-1 次的 Delete Min。最終，完整的 Delete Min 流程需經過$(n-1)log_2 n$ 次。

D. 分割 + 合併:

綜上所述，Heap Sort 不論何種情形，建立 Min Heap Tree n 個步驟，再加上合併的流程 $(n-1)log_2 n$ 個步驟，因此 Heap Sort 所有的流程需 (n) + $(n-1)log_2 n$ 個步驟，其時間複雜度為 O($nlog_2 n$)。

二、原理比較

1. Merge Sort

將一組為排序含有 N 個項目的數列，以「N/2」方式分割其長度，所以數列會分割成兩組，以「N/2」的商當作分割點的索引，左邊的數列範圍為原數列的起始索引到分割點的前一項索引；右邊的數列範圍為分割點的索引到原數列的尾端索引。接著，左右兩組分別繼續進行分割，直到數量的長度為 1 為止。接著，將分割後長度「1」的數列成對地合併並進行排序，直到合併成一組長度與最初數列一樣的數列，而該數列的數值則按照大小順序排序。見樹狀圖，最初在數列合併時會先從樹狀圖最底層長度「1」的數列開始合併。而每次在合併時會同時進行排序，首先，會有兩個指標指向欲合併的兩個數列起始數值；其次，兩個指標指向的數值進行比較，較小的數值將會作為合併後數列的起始值；接著，前一次有較小數值的數列的指標會指向該數列的下一個數值與另一個數列的指標所指向的數值繼續比較，較小的數值將會作為合併後數列的下一個數值，以此類推，直到某一個數列的指標超過數列範圍為止；最後，指標未指完的數列的數值再依序作為合併後數列的數值，上述過程為一次的合併與排序。完全分割的數列會不斷進行上述的合併與排序，直到合併成一組長度與最初數列一樣的數列。

2. Heap Sort

Heap Sort 的原理是先將數列排成堆積樹的形態，將位於樹根節點的數值與位於最後一個子節點的數值對調，再將對掉到最後一個子節點的數值剔除，作為新數列的第一順位，完成一次的 Heap Sort 排列。接著，因為原本堆積樹型態的數列已經剔除了位於最後一個節點的數值並且

原本位於樹根的數值也被移往最後一個節點的位置，因此原本堆積樹的型態已被破壞掉，此時將位於數根的數值向下依序與其較小的子節點比較，當該數值大於子節點時，兩數值則互換，互換後原位於數根的數值再繼續與更下層較小的子節點比較，直到比到最後一個節點為止，全部比較完後再將位於樹根節點的數值與位於最後一個子節點的數值對調，並將對調到最後一個子節點的數值剔除，排進新數列的第二順位。以此類堆，直到堆積樹型態的數列的數值完全被剔除，都被排到新數列中。而此時的新數列將會是一個經大小排序過的數列。

三、其他

Merge Sort 最重要的一個用途是外部排序，外部排序是指當排序的資料量無法直接在記憶體內進行排序，而必須使用到輔助記憶體，如硬碟。當資料量大到無法全部讀入主記憶體裡進行排序時，可以先讀入部分資料，例如針對以排序好的二個或二個以上的檔案，經過合併的方式，將其組合成一個大的且已排序好的檔案。

四、參考資料:

1. <使用資料結構 Python ，李淑馨著，深石數位科技 > 第九章 9.3.2 與 9.3.3 節
2. https://medium.com/appworks-school/%E5%88%9D%E5%AD%B8%E8%80%85%E5%AD%B8%E6%BC%94%E7%AE%97%E6%B3%95-%E6%8E%92%E5%BA%8F%E6%B3%95%E9%80%B2%E9%9A%8E-%E5%90%88%E4%BD%B5%E6%8E%92%E5%BA%8F%E6%B3%95-6252651c6f7e
3. http://notepad.yehyeh.net/Content/Algorithm/Sort/Heap/Heap.php