建議對演算法與數據結構已經有足夠熟悉的學員跳過此章節

主要原因是我們課程的設計核心為實際去模擬當你真實遇到該問題時會有什麼樣的想法，但講義內容其實已經有部分暗示解法，但實際遇到問題時並不會知道接下來的題目會用到解法的方向。

Recursion

recursion 最基礎的概念為我們寫了一個函數，且該函數內會呼叫自己。譬如說下列的程式就是一個計算 1~n 數字總和的 recursion：

```
void sum(int n){
   if (n == 1) {
       return 1;
   }
   return sum(n-1) + n;
}
```

在學習遞迴(recursion)時，主要需要考慮兩個重點：

1. 結束(邊界)條件:為了避免演算法無限執行，recursion必須在處理到資料量很小時停止，並直接進行處理。這通常是當資料只有一個元素時的情況。例如在排序的情境中，單一元素本身就已經是排序好的，因此如果我們在排序演算法中使用recursion，當資料只剩下一個元素時，就不需要再做recursion。
2. 回傳結果的處理:recursion 另一個重點是如何回傳正確的結果，當進入結束條件時，我們可以直接處理並回傳結果，如果還沒達到結束條件，則會在目前的函式中重複呼叫自己一次或多次。處理這部分的技巧在於「相信recursion會正確地回傳結果」。例如在計算 1 到 n 的總和這個例子中，我們要相信 sum(n-1) 回傳結果是對的，並由該回傳結果去算出sum(n)的結果。另一個例子是我們之後章節會題到的 merge sort，若我們用recursion來實作 sort，會先將輸入的array切成兩段，並對這兩個 array 用 recursion 的方式做排序，此時我們可以假設該recursion function會回傳一個 sort 好的 array，接下來我們要解決的問題就是，如何將這兩個 sorted 的 array 合併成一個新的 sorted array.

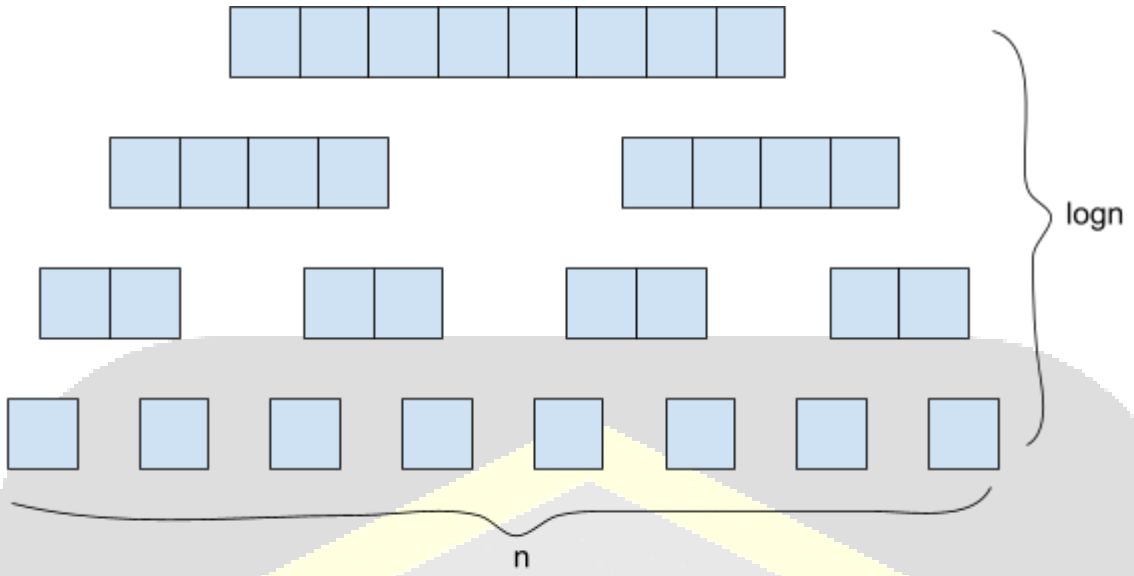
```
int[] sort(int[] array){
   array1 = first half of array
   array2 = last half of array
   array1 = sort(array1)
   array2 = sort(array2)
   // array1 and array2 is sorted
   // merge the array1 and array2 to a sorted array.
}
```

Sorting

Merge sort (with Divide and Conquer)

Merge sort 會運用到 divide and conquer 的技巧，也就是拆解+合併的運算過程，整個運算的過程靠遞迴進行。以下會分成 divide 和 conquer 兩部分講解。

時間複雜度為 $O(nlogn)$，因為總共會將 array 分成 $logn$ 層，且每層都會有 n 個元素，可以把他想像成用長乘寬算面積，因此時間複雜度為$O(logn) * O(n) = O(nlogn)$

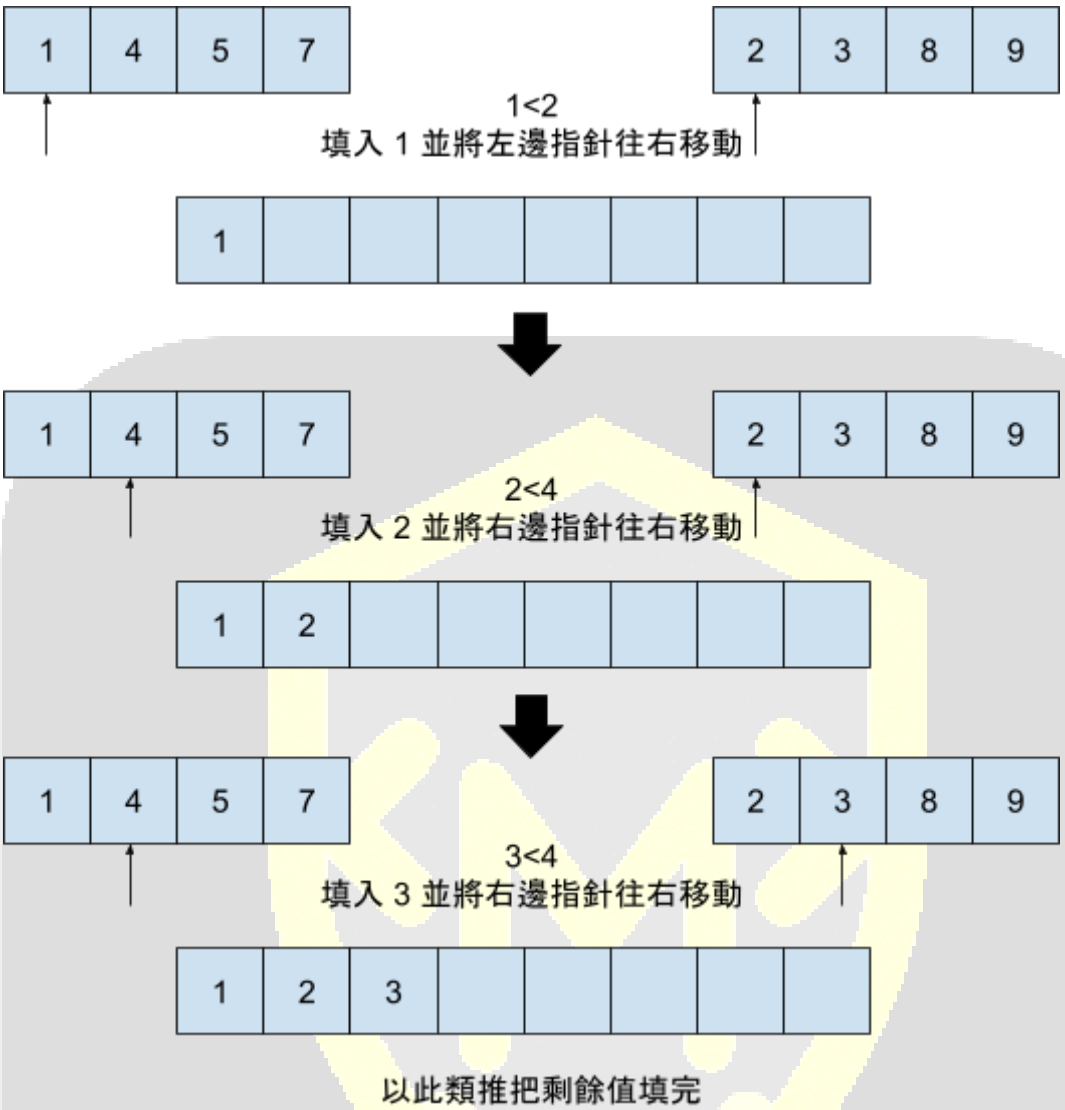


Divide

將原本 array 不斷拆成兩半再往下傳，在實作遞迴時將兩半陣列各自傳遞下去，直到拆到陣列長度為一時可開始做 conquer。

Conquer

由於左右兩半陣列都是 sorted array，因此可用以下方式合併兩個陣列。一開始將兩個指針各指在陣列的最左邊，然後比較兩邊的值大小，比較小的就將他拿出來放到新的陣列，然後把指針向右移動。當左右兩半陣列的指針都指向最後一格，即代表合併完成，可以回傳結果到上一層。
以下圖片為合併的示範，最後合併完可得到 `[1,2,3,4,5,7,8,9]`



Pseudo Implementation

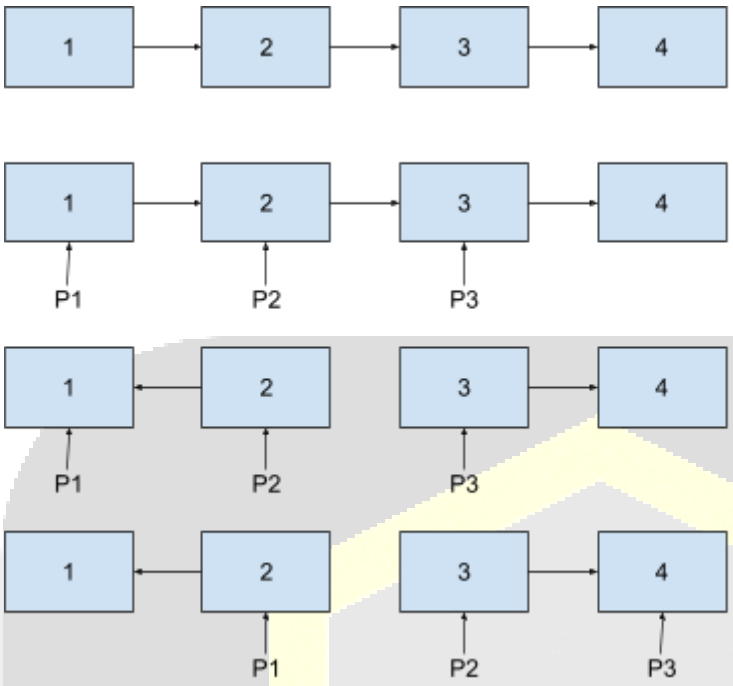
```
void mergeSort(int[] arr) {
    // recursion end condition
    if (arr.length <= 1) {
        return;
    }
    int mid = arr.length / 2;

    int[] left = new int[mid];
    int[] right = new int[arr.length - mid];

    for (int i = 0 ; i < mid ; i++) {
        left[i] = arr[i];
    }
    for (int i = mid ; i < arr.length ; i++) {
        right[i] = arr[i - mid];
```

```
    }

    mergeSort(left);
    mergeSort(right);

    merge(arr, left, right);
}

void merge(int[] arr, int[] left, int[] right) {
    int i = 0, j = 0, k = 0;

    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    while (i < left.length) {
        arr[k++] = left[i++];
    }
    while (j < right.length) {
        arr[k++] = right[j++];
    }
}
```

Linked list

Main point

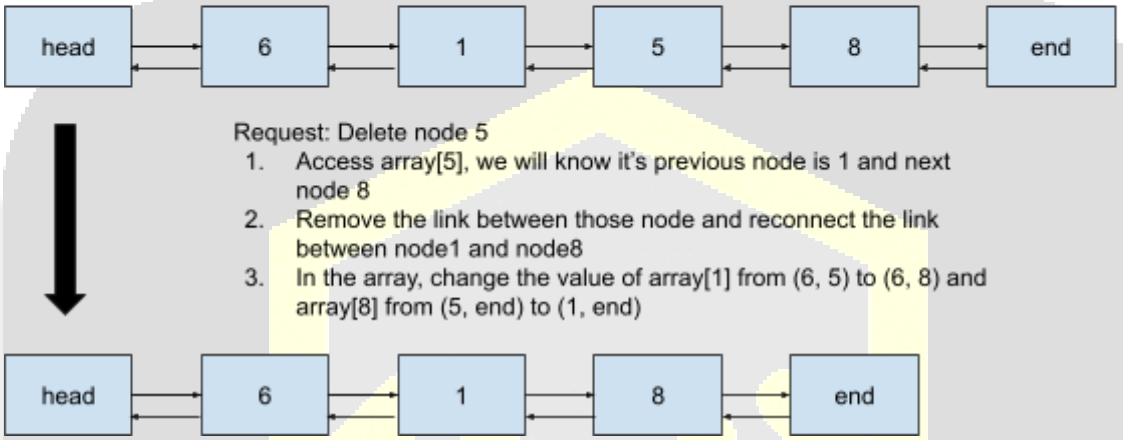
基本上 linked list 的題目所需要的操作就是：移除 link、增加 link和產生一個 pointer 指向 某個 node。而 linked list 最主要的重點是不管怎麼更改 link，都要確保每個 link 在刪掉之前他所指向的 node 都有別人指到，注意不能夠有孤兒 node 發生的狀況。
以下用反轉 linked list 的題目來示範如何確保每個 node 在操作時都還是會被指到



實作應用

很多人會好奇為甚麼解演算法題目的時候會用到 linked list，畢竟 array 就能夠儲存數值，使用 linked list 的優點何在。
在 array 要做 add 或 delete 一個值的操作需要 $O(n)$，反觀使用 linked list 配上 array 只需要 $O(1)$。以下用範例說明刪除一個 node 的操作
1. 藉由 array[5] 可得知 node 5 的前一個 node 是 1 下一個 node 是 8
2. 移除 node1 node5 和 node5 node8 之前的 link 然後重新連結 node1 和 node8
3. 在 array 裡將 array[1] 的值從(6,5) 修改為 (6,8)，array[8] 的值從(5,end) 修改為 (1,end)

0	1	2	3	4	5	6	7	8	9	Index
?	6	?	?	?	1	head	?	5	?	Value: pointer to previous
?	5	?	?	?	8	1	?	end	?	Value: pointer to next

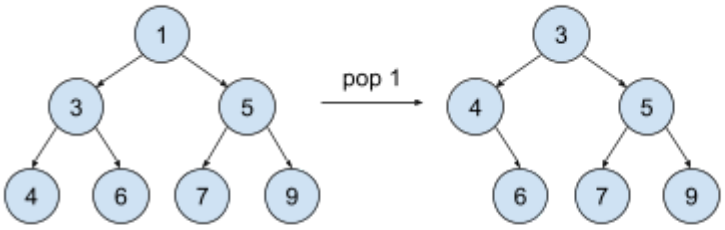


Priority queue

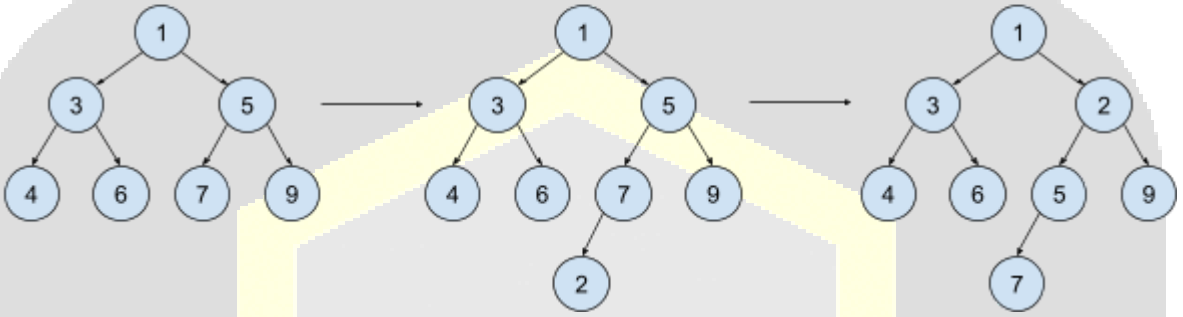
原理

Priority queue 通常就是拿來實作 min heap 和 max heap，在 min heap 中根的值一定是最小，且他的所有子樹也都會保持根是最小值。當我們今天要加一個新的點進到 min heap 裡，我們會先把它放在最下面（本來是 null 的位置），然後再藉由和 parent 比較大小一路調整網上位移。此時間複雜度為 $O(logn)$。每次從跟 pop 出最小值後需要將新的最小值移到根，並且需一路往下維護他的子樹（若左子樹數值比右子樹小就把左邊的點一上去然後往下維護左子樹）。Max heap 的做法跟 min heap 一樣只是根放的是最大值。

Pop root and adjust the queue



Add node 2 and adjust the queue



-