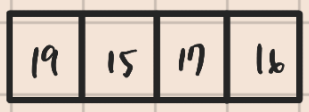
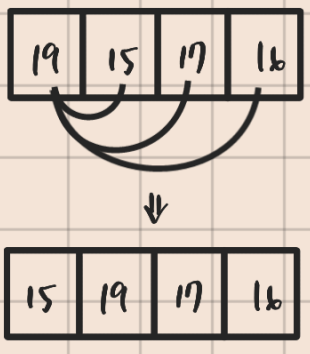
Selection Sort



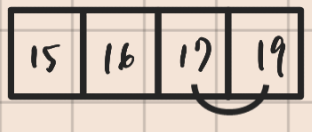
從 index:0 開始比較，找出 array 中最小的數字，並和 index 0 交換位置

1° 

⇒ 比完第1輪，15 為最小，將 15 和 19 (index:0) 交換位置

2° 

⇒ 第二輪從 index:1 開始比，找出最小值後交換位置，以此類推
(16、19 交換)

3° 

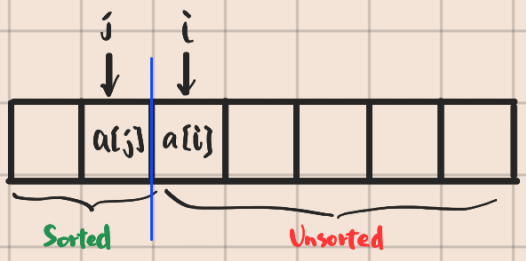
⇒ 3rd 輪，17<19 不用動，完成 *

SelectionSort.java × SortApp.java ×

```java
import java.util.Arrays;

public class SelectionSort {

    public static int[] selectionSort(int a[]){

        for (int i = 0; i < a.length; i++){

            int minIndex = i;

            for(int j = i+1; j < a.length; j++){

                if (a[minIndex] > a[j]) minIndex = j;
            }

            // swap min value to first position
            int holder = a[i];
            a[i] = a[minIndex];
            a[minIndex] = holder;
        }
```

```
        return a;
    }

}
```

Insertion Sort



i : unsorted 的第一个元素

j : sorted 的最大元素

1° `for ( int i=1 ; i< a.length ; i++ )` ← 從 2nd 元素開始，∵ index=0 視為已 sorted

`int element = a[i]` ← 先儲存第一个 unsorted 元素，待会比較用 (holder)

`int j = i-1` ← 將 j 指向最大的 sorted 值

`while ( j≥0 & a[j] > element )` ← 当 holder 大於 sorted的值

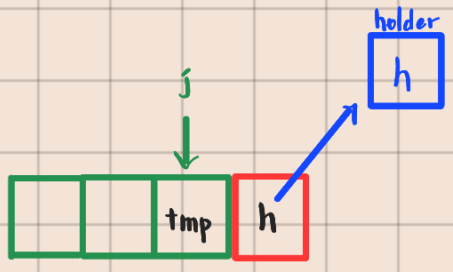
`a[j+1] = a[j]` ← 將 sorted 往右移

`j--` ← 將 pointer 往左，繼續比 (j=-1 表示已到 array 底，跳出迴圈)

`a[j+1] = element` ← 已達 array 底部，或 holder的值已找到合適位置

將 holder 值 插入

*. 程式概念图解



1° 先存 unsorted 的值 (holder, element)

將 pointer 指在 sorted 的最大值 (j=i-1)

將 h 与 tmp 比較，若 h 較小，

則將 tmp 右移

*. Note :

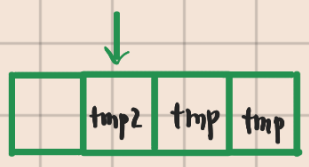
j : 指向 sorted 的最大值

i : 指向 unsorted 的 1st 元素

- ∵ index"0" 視為 sorted

∴ i 從 1 開始

repeat



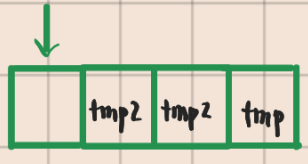
holder

繼續比較

- 若 h 小於 tmp2，

則將 tmp2 往右移，pointer 往左

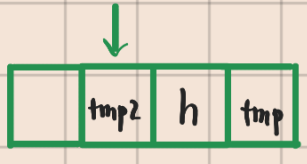
繼續重複，直到 pointer = index "0"
(j)



holder

- 若 h 大於 tmp2，

則將 h 寫在 tmp2 右邊一格 ⇒ 完成
(j+1)



holder

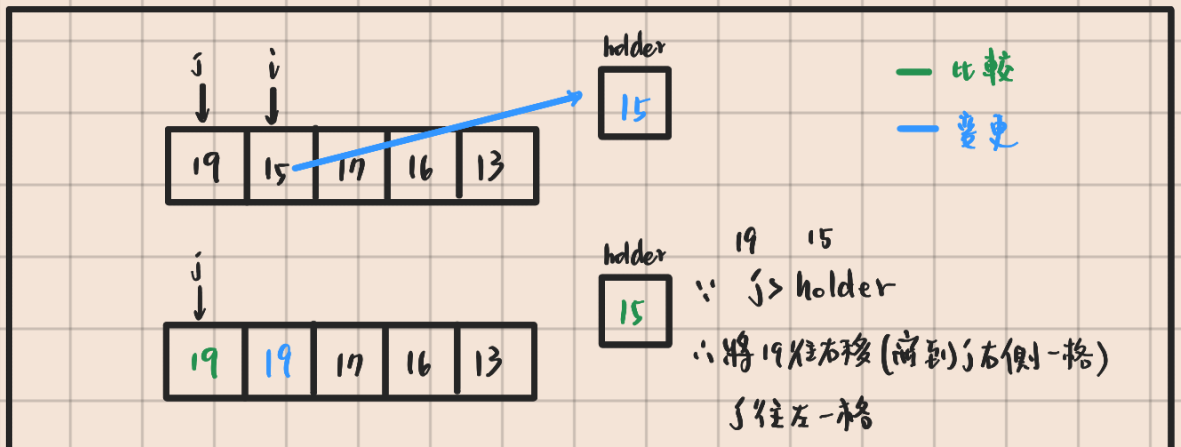
i++，繼續處理

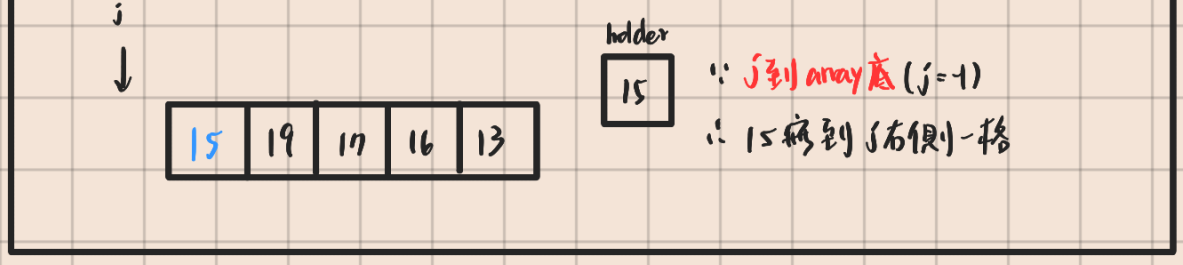
下一个 unsorted value □

*心得：比較完，若 holder 的值比較小，holder 的值不要急著寫入

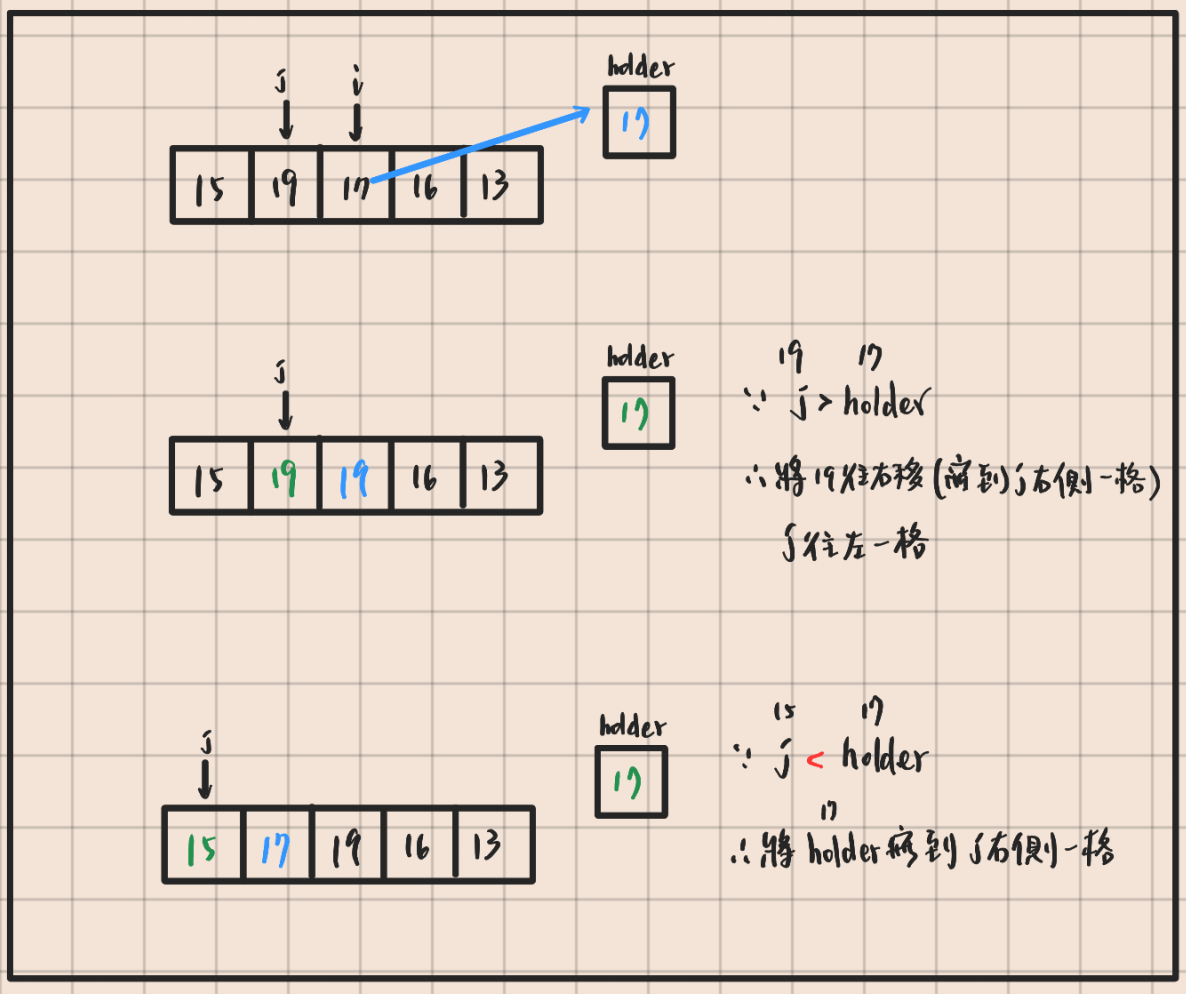
先把 tmp 的值往右移就好，

只有：① holder 的值比 tmp 大
② 已經比到 array 底部
} ⇒ 才會寫入 holder 值

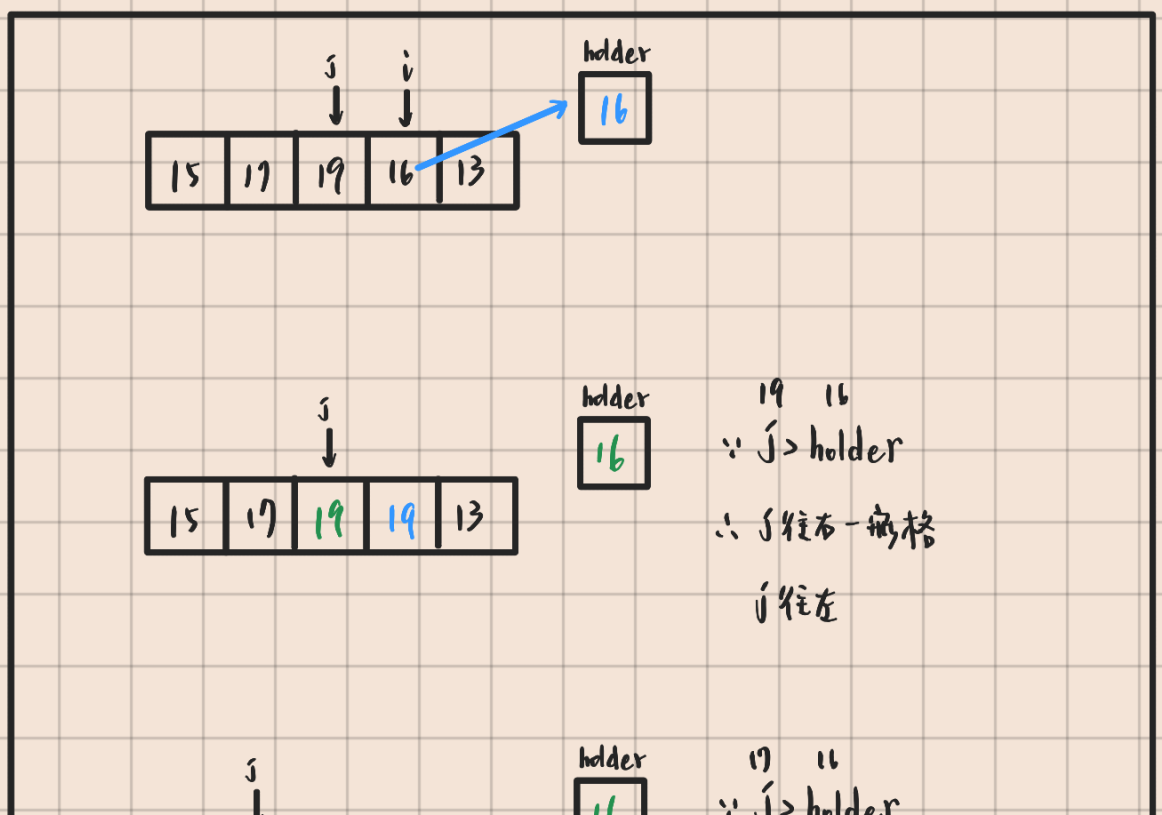


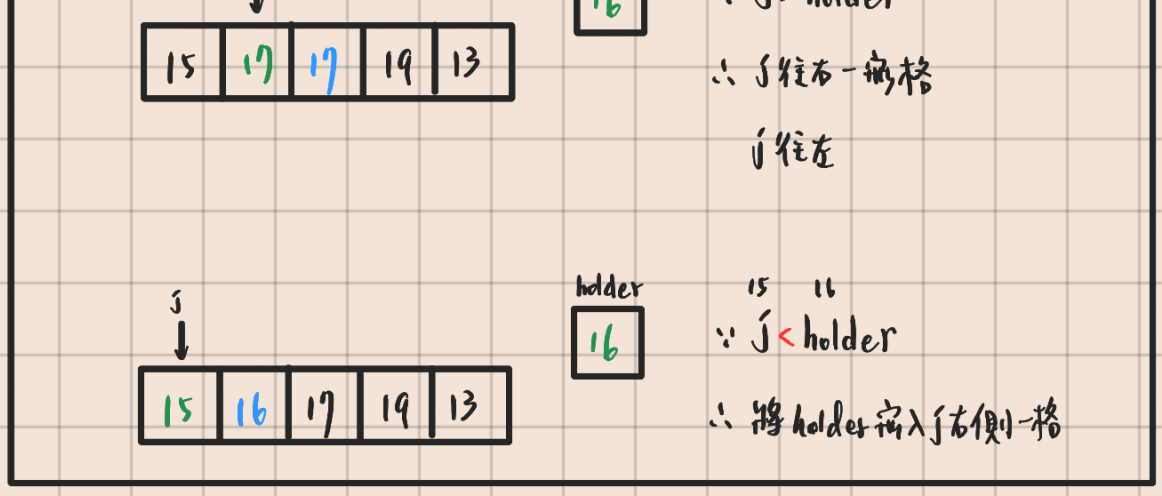

j
15 19 17 16 13
holder
15
∵ j到array底 (j=-1)
∴ 15搬到j右側一格

i = 1


j i
15 19 17 16 13
holder
17
j
15 19 19 16 13
holder
17
19 17
∵ j > holder
∴ 將19往右移 (前到j右側一格)
j往左一格
j
15 17 19 16 13
holder
17
15 17
∵ j < holder
17
∴ 將holder搬到j右側一格

i = 2


j i
15 17 19 16 13
holder
16
j
15 17 19 19 13
holder
16
19 16
∵ j > holder
∴ j往右一格
j往左
j
holder
17 16
∵ j > holder



i=4 以此類推

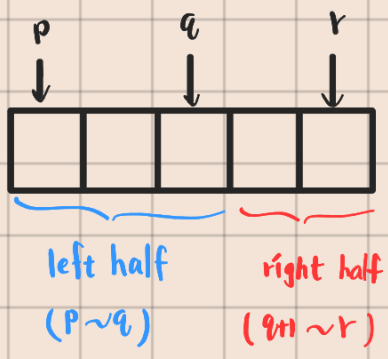
```java
package SortAlgorithm;

public class InsertionSort {

    public static int[] insertionSort(int a[]) {

        for (int i = 1; i < a.length; i++){  // Caution : starts from index "1"
            int element = a[i];
            int j = i-1;

            while (j >= 0 && element < a[j]){
                a[j+1] = a[j];
                j--;
            }
            a[j+1] = element;
        }

        return a;
    }
}
```

Merge Sort

*. Merge Sort Pseudocode

P=q 時(只有1元素)才不繼續拆解

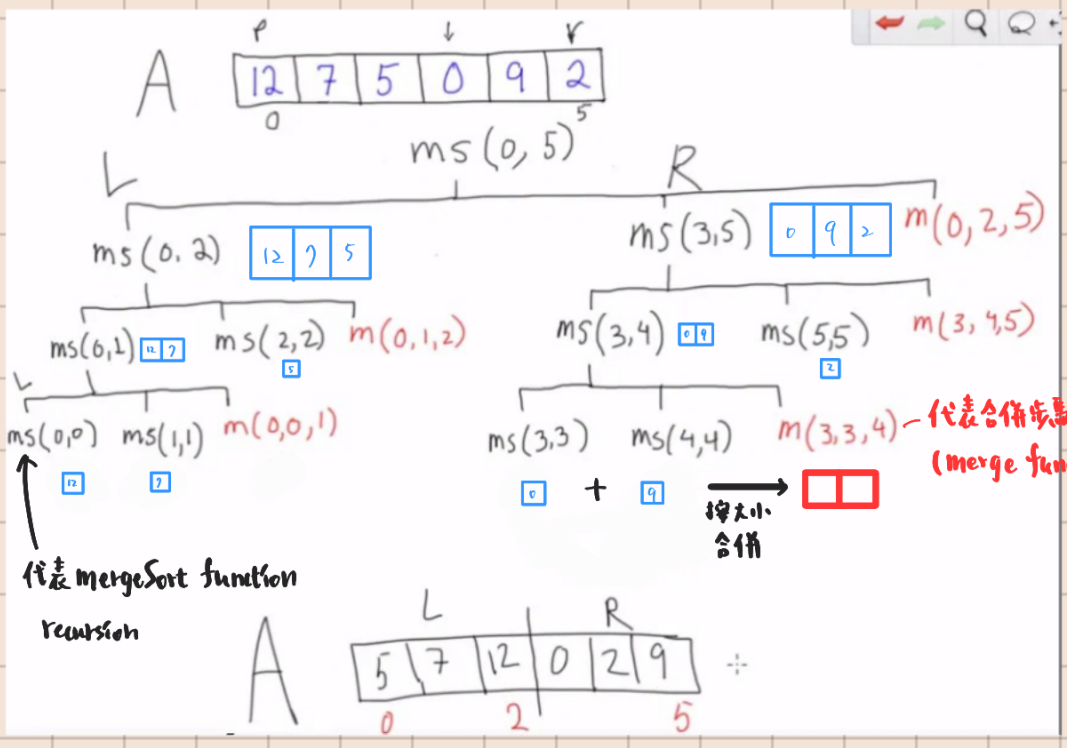
```
mergeSort(A, p, r) {
    if (p < r) {
        q = [(p + r)/2]
拆為左半部 ⟶ mergeSort(A, p, q)
拆為右半部 ⟶ mergeSort(A, q + 1, r)
合併 ⟶ merge(A, p, q, r)
    }
}
```



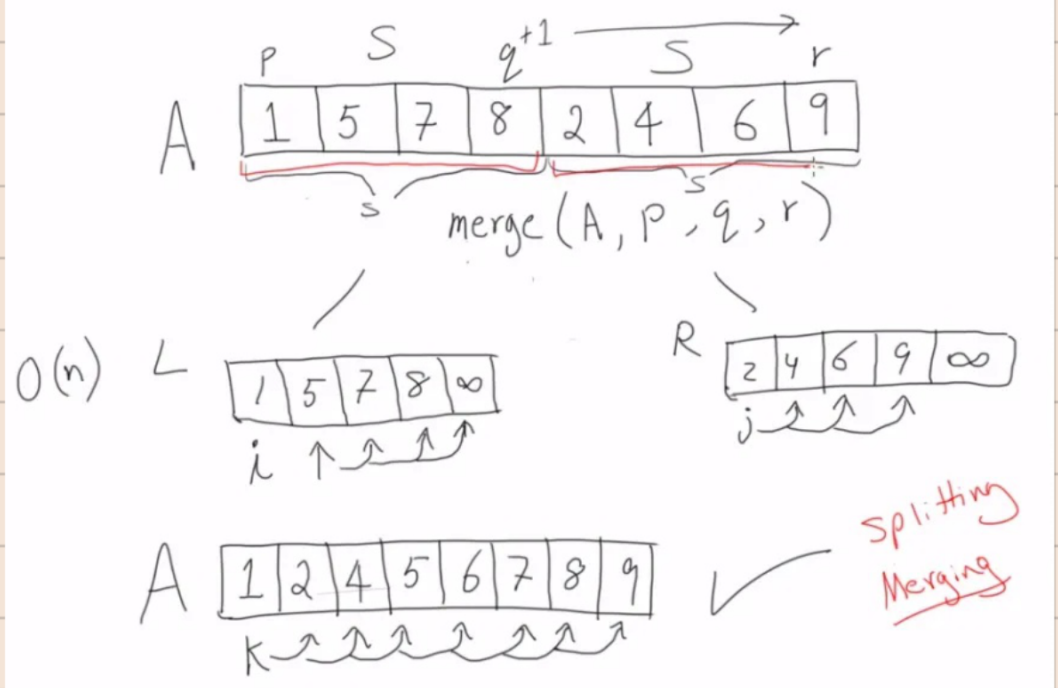
$O(n) = \text{merge 次数} \times \text{merge function}$

$= \frac{\log_2 n}{\text{一直分2半}} \times \frac{n}{\text{1个for loop}} = n \log n$

*. Pseudocode 图解



*. merge() function 图解



MergeSort.java | MergeSortAnswer.java | SortApp.java

```java
public class MergeSort {

    public static void sort(int inputArray[]){
        sort(inputArray, start: 0, end: inputArray.length-1);
    }

    public static void sort(int inputArray[], int start, int end){
        if(end <= start){
            return; // We're done traversing the array
        }
```

只是將 input array 加上 p, r (start 和 end)

当 start < end 时，就继续分左右半，等于 start ≥ end 时 return

```java
        int mid = (start + end)/2;
        sort(inputArray, start, mid); // sort left half
        sort(inputArray, start: mid+1, end); // sort right half
        merge(inputArray, start, mid, end); // merge sorted results into the inputArray
    }

    public static void merge(int A[], int start, int mid, int end) {

        // init l, r array (+1 for largest value)
        int mergeArraySize = end - start +1;
        int lArraySize = (mid - start + 1) + 1;
        int rArraySize = (end - (mid+1) +1) + 1;
        int[] lArray = new int[lArraySize];
        int[] rArray = new int[rArraySize];
        // write array to L, R
        for (int i = 0; i < lArraySize - 1; i++){
            lArray[i] = A[start + i];
        }

        for (int i = 0; i < rArraySize - 1; i++){
            rArray[i] = A[(mid+1) + i];
        }

        // Add largest value to the end of l, r
        lArray[lArraySize - 1] = Integer.MAX_VALUE;
        rArray[rArraySize - 1] = Integer.MAX_VALUE;

        // merge
        int lPointer = 0;
        int rPointer = 0;

        for(int k = 0; k < mergeArraySize; k++){ // array A pointer "k"

            // move smaller value to array A, & move pointer in l, r
            if (lArray[lPointer] < rArray[rPointer]){
                A[start + k] = lArray[lPointer];
                lPointer++;
            } else { // r smaller or equal
                A[start + k] = rArray[rPointer];
                rPointer++;
            }
        }

        // Debugging purpose
        //System.out.println("Merge(" + start + "," + mid + "," + end + ")");
        //System.out.println(Arrays.toString(lArray));
        //System.out.println(Arrays.toString(rArray));
        //System.out.println(Arrays.toString(A));
    }

    public static void main(String[] args) {

        // Test case 1 :
        merge(new int[] {1,5,7,8,2,4,6,9}, start: 0, mid: 3, end: 7);

        // Test case 2 :
        merge(new int[] {1,2,3,4,5,6,7} , start: 0, mid: 3, end: 6); //{1,5,7,8,10,2,4,6,9}

        // Test case 3 :
        int[] a = new int[] {12,7,5,0,9,2}; //{1,5,7,8,10,2,4,6,9}
        sort(a);
        System.out.println(Arrays.toString(a));
    }

}
```

排序+合併

init l,r

計算合併後的array大小

不可用 A.length/2，∵不論recursion幾次，這裡傳入的A指的都是原始矩陣！(含全部數值的array)

存∞用

copy data to l,r

用Pseudocode圖解中，右半{0,9}&{2}合併來理解

Add ∞ to l,r

有∞，不用考慮其中一側矩陣(L,R)先清空的情況

已事先知到合併後的array size，∴k滿代表已合併排序完畢

Sort l,r & write A[]

加上start，才不會將右側排序完的array寫至A的左側

討論：

- Q：傳值 or 傳參考？

  method(int i) ⟶ i 不會被修改

  method(int i[]) ⟶ array i 會被修改

  method(Object O) ⟶ Object O 會被修改

A：Java 中，若是 primitive variable (如 int, double…)，是傳值，原值不會改變

若是 array 或 object 這種 reference variable，會將其地址傳入，因此原值有可能改變

原本以為：~~用 "new" 宣告的都會改變（動態配置、位址）~~ ← String 不是 primitive，∴是傳地址

但傳入 function 不一定改變

結論：Java 只有傳"值"，都是傳我值

只是 primitive variable 傳的值是"我值"（8種：byte short int long float double boolean char）

而 String, Array, Classes 傳的值是"地址"

至於傳入地址後，原值會不會改變，要看更改的方式！

(primitive 傳值是一定不會改變)

↓ 以非 primitive 的 String 測試

MyClass.java

```java
package com.logan.ds;

import java.util.Arrays;

public class MyClass {
    public static void main(String[] args) {

        String s1 = new String( original: "String is non-primitive, should be changed in method");
        method(s1);
        System.out.println(s1);   // > String is non-primitive...

    }

    public static void method(String s1) {
        String s2 = new String( original: "It change!");
        s1 = s2;
    }
}
```

非 primitive，傳入 s1 的位址

用 s2 的新 addr 蓋過原本 s1 的 addr，只有 function 內 s1 改變，也沒有 return s1 ∴原值不變

若是寫 s1.method() 去改寫 s1 所指向的我值，s1 就會改變!!!

Ref：StackOverflow，Is Java "pass-by-reference" or "pass-by-value"?

* 總而言之，Primitive variable 一定不變！

String, Array, Object 要看修改方式！