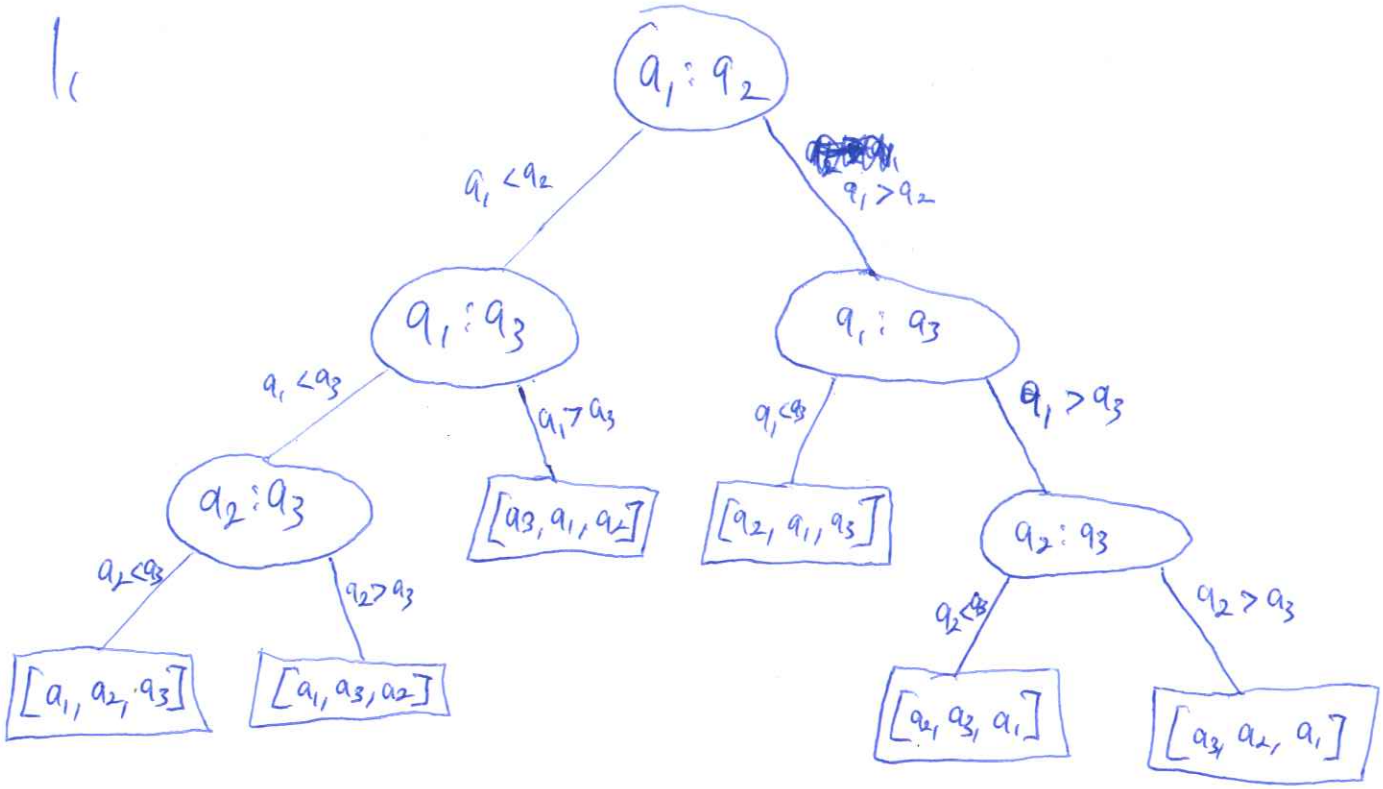
1.


$a_1 : a_2$
$a_1 < a_2$
$a_1 > a_2$
$a_1 : a_3$
$a_1 : a_3$
$a_1 < a_3$
$a_1 > a_3$
$a_1 < a_3$
$a_1 > a_3$
$a_2 : a_3$
$[a_3, a_1, a_2]$
$[a_2, a_1, a_3]$
$a_2 : a_3$
$a_2 < a_3$
$a_2 > a_3$
$a_2 < a_3$
$a_2 > a_3$
$[a_1, a_2, a_3]$
$[a_1, a_3, a_2]$
$[a_2, a_3, a_1]$
$[a_3, a_2, a_1]$

2. (a) let the input array be $A[a_1, a_2, a_3, \ldots a_n]$ and assume $a_1 = a_2 = a_3 = \cdots = a_n = k$

When this array is processed through counting sort algorithm, the auxiliary storage array will be

$$c[i] = \begin{cases} 0 & \text{for } i \neq k \\ n & \text{for } i = k \end{cases}$$

The first iteration of the loop populating $B[]$ array will be $B[C[A[n]]] = A[n]$

it evaluates to $B[n] = A[n] = k$ ——— (1)

the second iteration will be

$B[c[A[n-1]]] = A[n-1]$

or $B[n-1] = A[n-1] = k$ ——— (11)

From (1) and (11) we can generalize that for the ith iteration $B[i] = A[i] = k$.

Even when the input array has only one distinct key the algorithm populates the result array based on the sequence of keys in input array.

The last loop which re-arranges the array is

```
for j = n to 1
    B[c[A[j]]] = A[j]
    C[A[j]]--;
```

The element $A[j]$ will be arranged in the same order as in the input array if the key is of the same value for two elements. so the order is preserved and counting sort is stable, we can conclude.

(5) (i) The conditional check deciding whether an element needs to be repositioned is:

```
while(i > 0 and A[i] > key)
```

for the key to be moved to the left of $A[i]$ it has to be less than $A[i]$ and if $A[i] =$ key, the element $A[i]$ will not be repositioned

with respect to key. So the order of equal keys are preserved in the output. This implies insertion sort is stable.

(ii) In merge subroutine the condition check for populating output array is;

```
if L[i] ≤ R[i]
    A[k] = L[i]
else
    A[k] = R[i]
```

So the value from right array will be placed before the value in left array in the output only if it is greater. In case of equal keys the key from left subproblem will be placed before the key from right subproblem. So the order will be preserved and merge sort is stable.

(iii) Quicksort is not stable because the seqeuence of occurance of items with same key is changed during the swap operation in partition() subroutine. Consider the following seqeuence of numbers;

6, 4, 2, 2, 5

To distinguish between two 2's we mark them as follows; 6, 4, 2_a, 2_b, 5 (2_a appears before 2_b in original input)

If we choose 5 (the last element) as pivot,

After partition the array will be as follows:

$2_b, 4, 2_a, 5, 6$

And finally the sorted array will be

$2_b, 2_a, 4, 5, 6$

Hence 2_b comes before 2_a shows that quicksort is not stable.

3.

646	196	619	920
920	167	920	541
619	541	541	661
853	582 →	646 →	582
864	646	853	853
541 →	619	167	864
196	678	661	646
582	661	864	196
167	853	678	167
678	864	582	678
661	920	196	619
↑	↑	↑	

Clearly these are not sorted. To fix, partition the numbers into "bins" by their digits, and only sort less significant digits from numbers in the same bin.

646	196	167
920	167	196
619	541	541
853	582	582
864 →	646 →	619
541	619	646
196	678	661
582	661	853
167	853	864
678	864	920
661	920	
↑		

Each bin has 1 element after sorting 2 digits so we are done.

4.

```
median ( A, B, sa, ea, sb, eb)
{
    ma = ⌊(sa + ea)/2⌋ ; // median of A
    mb = ⌊(sb + eb)/2⌋ ; // median of B
    if (A.length == 1 and B.length == 1)
        return (A[0] + B[0])/2 ;
    else if (A.length == 2 and B.length == 2)
        return ( max( A[0], B[0])+
                 min( A[1], B[1]))/2 ;
    else if (A[ma] > B[mb])
        return median ( A, B, sa, ma-1, mb+1,
                        eb)
    else if (A[ma] < B[mb])
        return median(A, B, ma+1, ea, sb, mb-1)
    else    return A[ma] ; // case: ma = mb
```