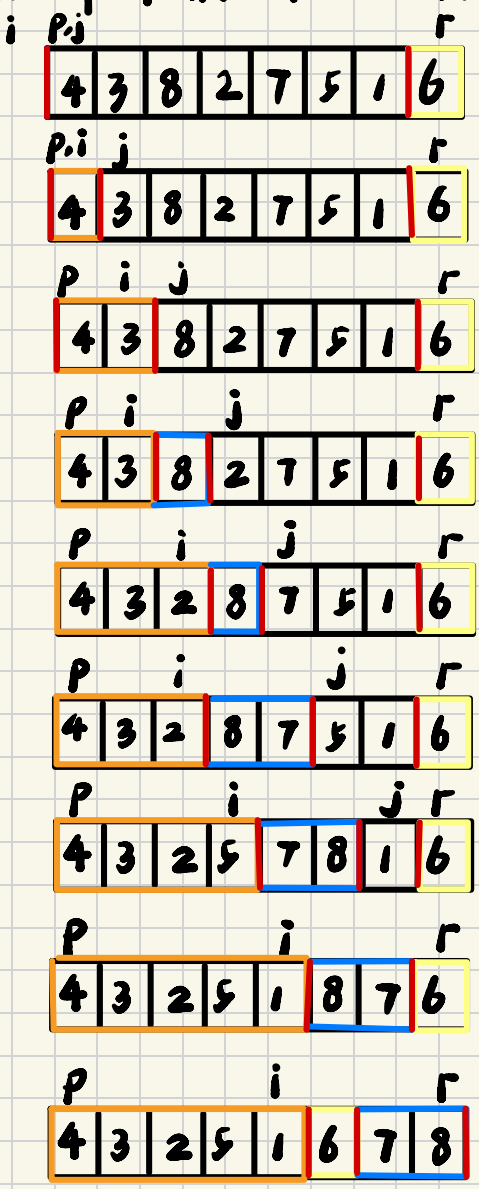
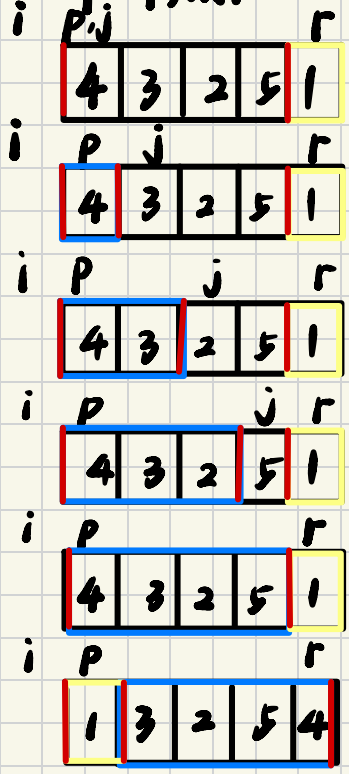
Question 5.1

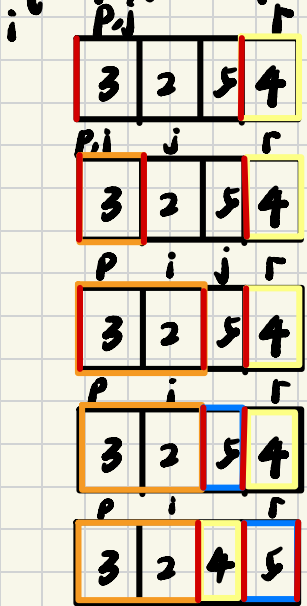
(1) q = PARTITION(A, 1, 8)



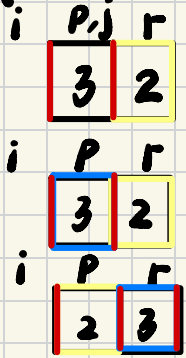
q = PARTITION(A, 1, 5)



q = PARTITION(A, 2, 5)



q = PARTITION (A, 2, 3)



q = PARTITION (A, 7, 8)

QUICKSORT (A, 1, 8)

QUICKSORT (A, 1, 5)

QUICKSORT (A, 2, 5)

QUICKSORT (A, 2, 3)

QUICKSORT (A, 7, 8)

Question 5.2

Loop invarient : every time it calls PARTITION (A, p, r) then elements in $A[p \cdots q-1]$ is not larger than $A[q]$, elements in $A[q+1 \cdots r]$ is larger than $A[q]$.

Initialization: the first time when PARTITION (A, p, r) is called, for PARTITION's correctness, it ensures that elements in $A[p \cdots q-1] \leq A[q]$, elements in $A[q+1, r] > A[q]$ And the loop invarient is established.

Maintanence: After PARTITION(A,p,r),
recursion will call QUICKSORT(A,p,q-1)
and QUICKSORT(A,q+1,r).
During QUICKSORT(A,p,q-1), we won't change
the order of A[q] and A[q+1...r], and we
can get $q_1' = $ PARTITION(A,p,q-1).
elements in $A[p...q_1'-1] \leq A[q_1']$
elements in $A[q_1'+1...q-1] > A[q_1']$
For QUICKSORT(A,q+1,r) is alike.
Then the loop invarient holds.

Termination:
When the size of subarray is 0 or 1, the subarray
is sorted naturally. No more PARTITION is called.
The loop invarient still holds.

Question 5.3
For every QUICKSORT(A,p,r), q=PARTITION(A,p,r)
Then we need to sort two subarray, with size of
0 and (r-p). (It means another (n-1) question)
The each PARTITION takes $\Theta(n)$, the sort of
array with size 0 takes $\Theta(1)$

Then, $T(n) = T(n-1) + \Theta(n) + \Theta(1)$

$T(n-1) = T(n-2) + \Theta(n-1) + \Theta(1)$

$\vdots$

$T(2) = T(1) + \Theta(2) + \Theta(1)$

Thus, $T(n) = \Theta(n^2)$

Question 5.4

(1) When all n elements have the same value, in every time of for-loop in PARTITION, $i = i+1$ then in final $i = r-1$, and $q = i+1 = r$

$q = r$

(2) Every PARTITION(A, p, r) takes $\Theta(n)$

After PARTITION, we need to sort two subarrays, with the size of 0 and $(r-p)$

We find it's the worst case, and the same as Question 5.3.

Then $T(n) = \Theta(n^2)$

Question 5.5

QUICKSORT:

```
public static void quickSort(int[] a,int p,int r){
    if(p<r){
        int q[]=partition(a,p,r);
        quickSort(a,p, r: q[0]-1);
        quickSort(a, p: q[1]+1,r);
    }
}
```

PARTITION:

```java
public static int[] partition(int[] a ,int p,int j){
    int x = a[j];
    int i = p - 1;
    int k = j;
    int pt = p;

    while (pt < k) {
        if (a[pt] < x) {
            i++;
            int t = a[pt];
            a[pt] = a[i];
            a[i] = t;
            pt++;
        } else if (a[pt] > x) {
            k--;
            int t = a[pt];
            a[pt] = a[k];
            a[k] = t;
        } else {
            pt++;
        }
    }

    int tempt = a[k];
    a[k] = a[j];
    a[j] = tempt;
    int q=i+1;
    int t=k;
    return new int[] { q, t };
}
```

Idea: use one more pointer to locate the elements that are larger than x.
Then we can ensure that
$A[p \cdots i] < x$ $A[i+1 \cdots k] = x$
$A[k+1 \cdots j-1] > x$

Argument:
Everytime we call PARTITION, we make all elements equal to x in their right place. Then we sort elements that are larger than x or smaller than x. By recursion we can get sorted array.

Runtime of QUICKSORT on the input from Q5.4 is $\Theta(n)$.