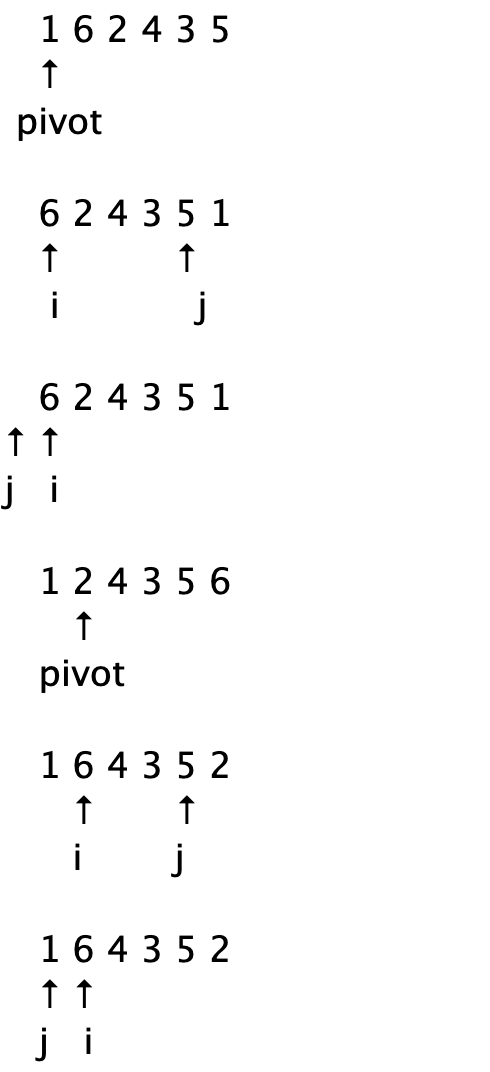
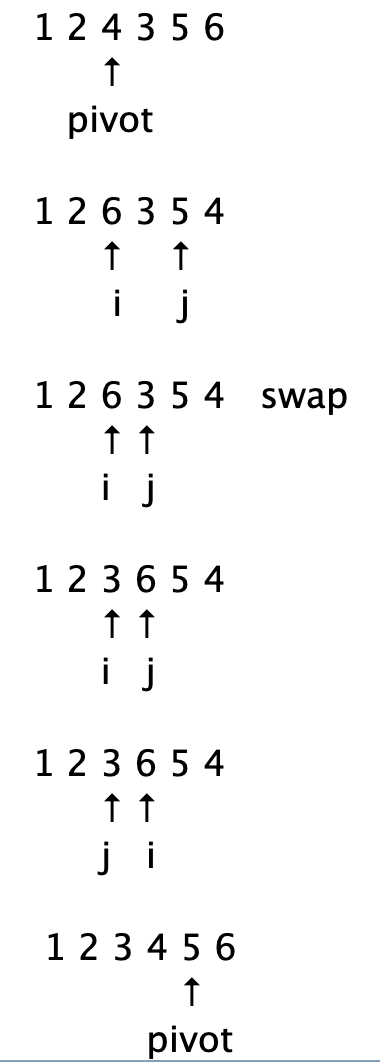
**LAB5**

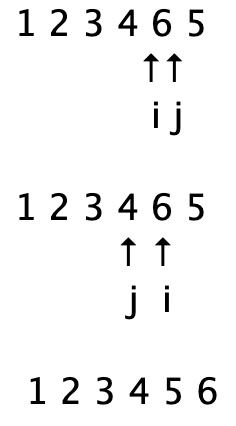
**Jimmy Palma**

**610756**

1. **Show all steps of In-Place QuickSort in sorting the array [1, 6, 2, 4, 3, 5] when doing first partition. Use leftmost values as pivots.**







1. **In our average case analysis of QuickSort, we defined a *good self-call* to be one in which the pivot *x* is chosen so that number of elements < x is less than 3n/4, and also the number of elements > x is less than 3n/4. We call an x with these properties a *good pivot.* When n is a power of 2, it is not hard to see that at least half of the elements in an n-element array could be used as a good pivot (exactly half if there are no duplicates). For this exercise, you will verify this property for the array A = [5, 1, 4, 3, 6, 2, 7, 1, 3] (here, n = 9). Note: For this analysis, use the version of QuickSort in which partitioning produces 3 subsequences *L, E, R* of the input sequence *S.*** 
   1. **Which x in A are good pivots? In other words, which values x in A satisfy:** 
      1. **the number of elements < x is less than 3n/4, and also**
      2. **the number of elements > x is less than 3n/4**

Good self call = 3n/4 -> 3(9)/4 = 6,75

Pivot 2: #L = 2, #G = 6

Pivot 3: #L = 3, #G = 4

Pivot 4: #L = 5, #G = 3

Pivot 5: #L = 6, #G = 2

Good pivots = 2, 3, 3, 4, 5

* 1. **Is it true that at least half the elements of A are good pivots?**

Yes, 5/9 are goods pivots

1. ***Interview Question.* Give an o(n) (“little-oh”) algorithm for determining whether a sorted array A of distinct integers contains an element m for which A[m] = m. You must also provide a proof that your algorithm runs in o(n) time.**

Algorithm find

Input: sorted array

Output: boolean

If arr.size == 1 and arr[mid] <> mid then

Return false

Else

return true

Mid = (0 + arr.size)/2

If arr[mid] == mid then

Return true

Else

Return Find(arr, 0, mid – 1)

Return Find(arr, mid+1, arr.size)

/////////// code /////////////

public static boolean find(int arr[], int first, int last){

if (last>=first){

if(arr.length < 2) {

if(arr[0] == 0) {

return true;

} else {

return false;

}

}

int mid = (first + last)/2;

if (arr[mid] == mid){

return true;

}

return *find*(arr, first, mid-1) || *find*(arr, mid+1, last);

}

return false;

}

1. **Prove the following recursive factorial algorithm is correct.**

**Algorithm recursiveFactorial(n)**

***Input*: A non-negative integer n**

***Output*: n!**  
 **if (n = 0 || n = 1) then**

**return 1**  
 **return n \* recursiveFactorial(n-1)**

* 1. Base case: is true, algorithm has a base case
  2. n = 1 || n=0 return 1, is true
  3. Assuming algorithm is correct for array < n, when the length array is n, each recursive call reduce the length until it can achieve the base case

1. **Review of SubsetSum Problem: Given a set S = {s0, s1,s2, ..., sn-1} of positive integers and a non-negative integer k, find a subset T of S so that the sum of the integers in T equals k or indicate no such subset can be found.**

**We have already seen a brute force solution to this problem in an earlier lab. In this exercise, you are going to come up with a recursive solution for SubsetSum. Write the pseudo code for your algorithm.**

Algorithm SubsetSum(S, k)

Input: S positive interger

Output: S₀, sum(S₀) = k

If S.length == 0 then

If k == 0 then

Return {}

Else if k = S₀ then

Return S₀

Else

Return null

S₀ <- S.removeLast()

Return SubsetSum (S₀, k)