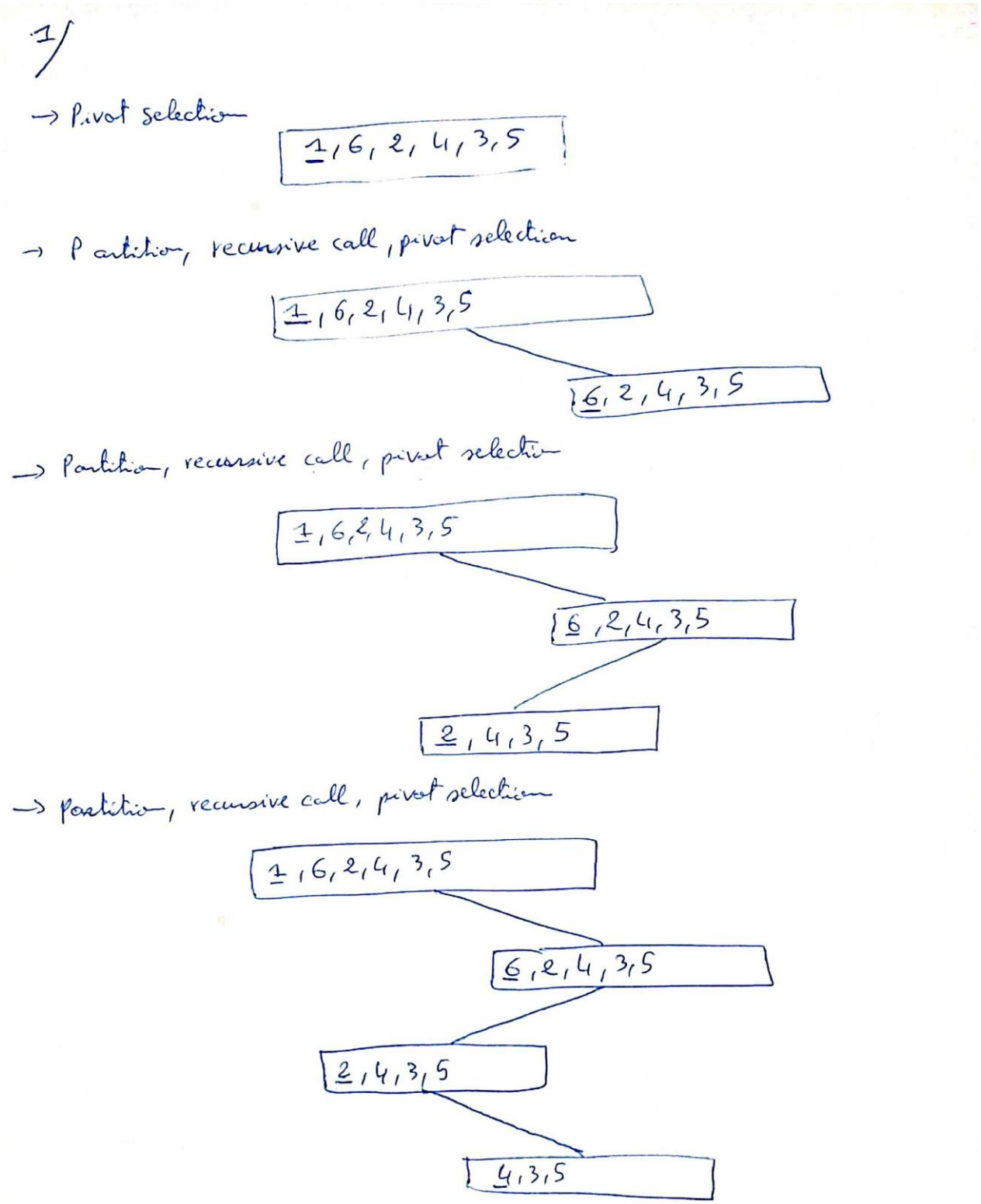
Lab 5

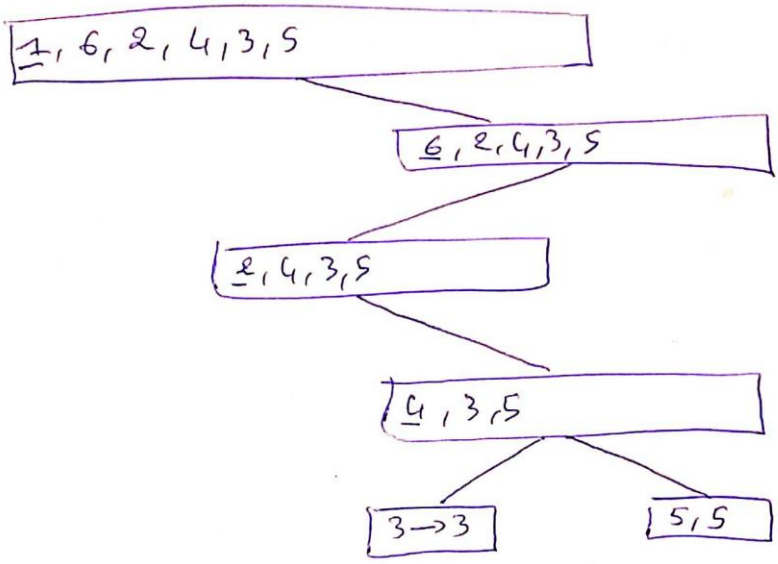
Houssam Eddine ATIF

N# 610165

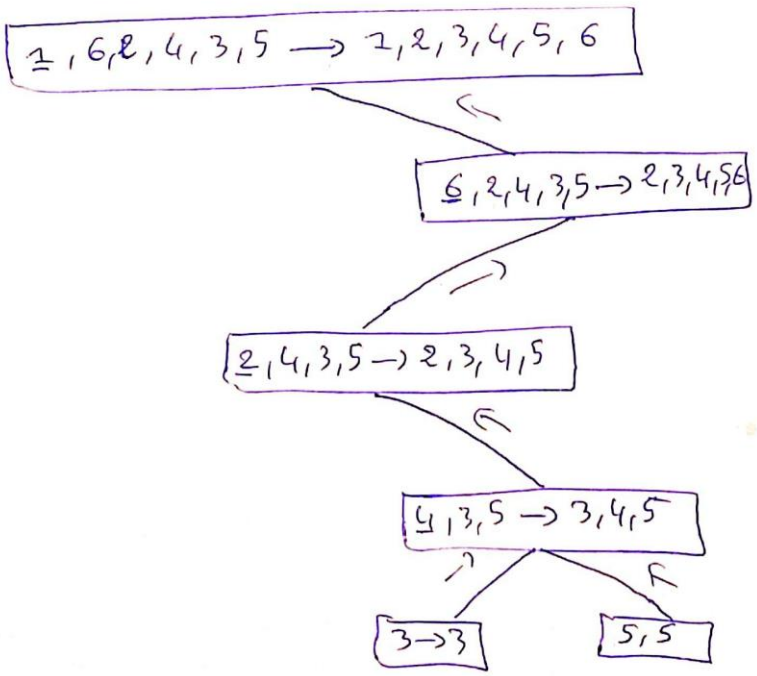
1/


1/
→ Pivot selection
1, 6, 2, 4, 3, 5
→ Partition, recursive call, pivot selection
1, 6, 2, 4, 3, 5
6, 2, 4, 3, 5
→ Partition, recursive call, pivot selection
1, 6, 2, 4, 3, 5
6, 2, 4, 3, 5
2, 4, 3, 5
→ Partition, recursive call, pivot selection
1, 6, 2, 4, 3, 5
6, 2, 4, 3, 5
2, 4, 3, 5
4, 3, 5

→ Partition, base case



→ join, join, join, join



2/

2/ -> Select the pivot (leftmost value) and swap with rightmost element

1, 6, 2, 4, 3, 5

↓

5, 6, 2, 4, 3, 1

i → ... ← j

-> i moves right as long as it points to values < 1

-> i is stuck at 5. Now j moves to the left as long as it points to a number > 1

5, 6, 2, 4, 3, 1

i → ... ← j

-> j is now stuck at 5 and since i stuck also at 5 we will swap 5 with the pivot 1 and do a recursive call for the array that contain the other elements except the pivot(1)

1, 6, 2, 4, 3, 5

↓ Recursive call

-> Select the pivot (leftmost) and swap it with the rightmost

1, 5, 2, 4, 3, 6

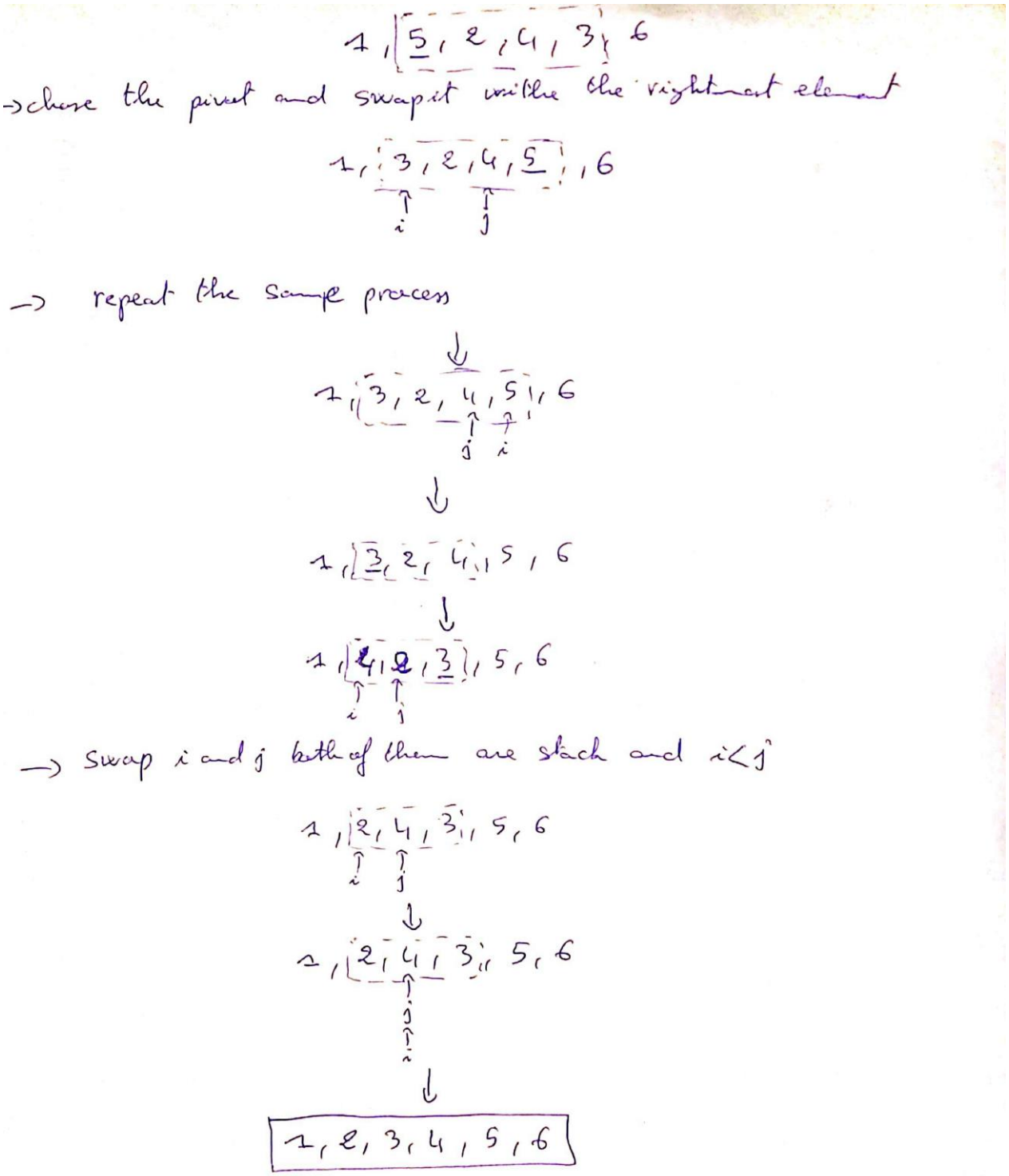
i → ... ← j

-> j moves right as long as values < 6

1, 5, 2, 4, 3, 6

... j → i

-> i > j -> Stop and swap i with the pivot and recursive call for the other elements.



-> choose the pivot and swap it with the rightmost element

-> repeat the same process

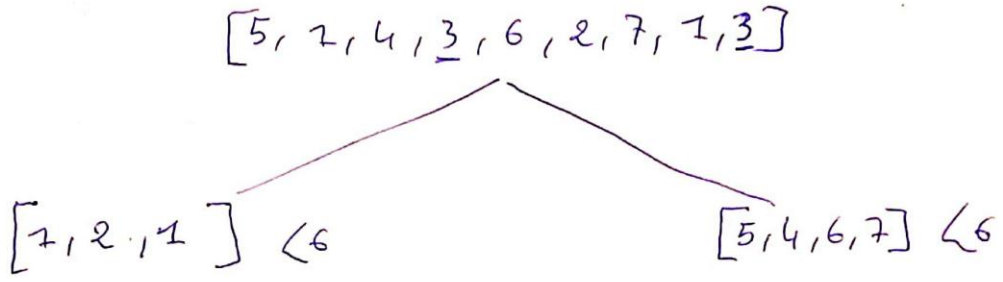
-> swap i and j both of them are stuck and i<j

3/

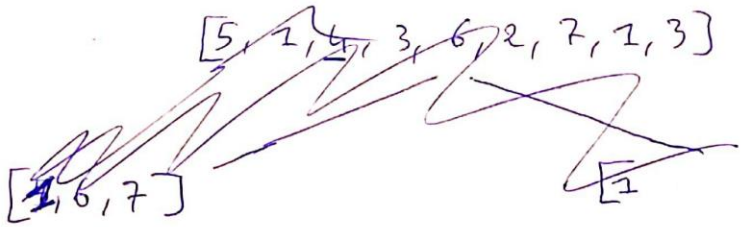
3/

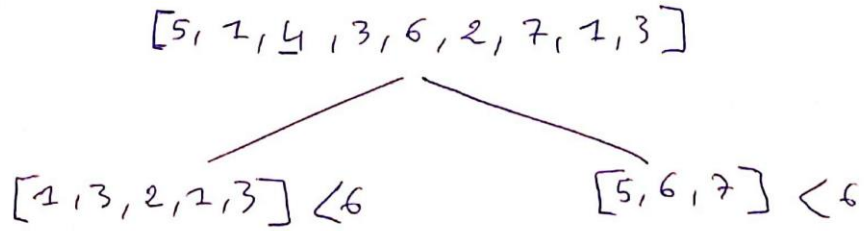
$A = [5, 1, 4, 3, 6, 2, 7, 1, 3]$ / $\frac{3n}{4} = \frac{3 \times 9}{4} \simeq 6$

a/ the good pivots are :

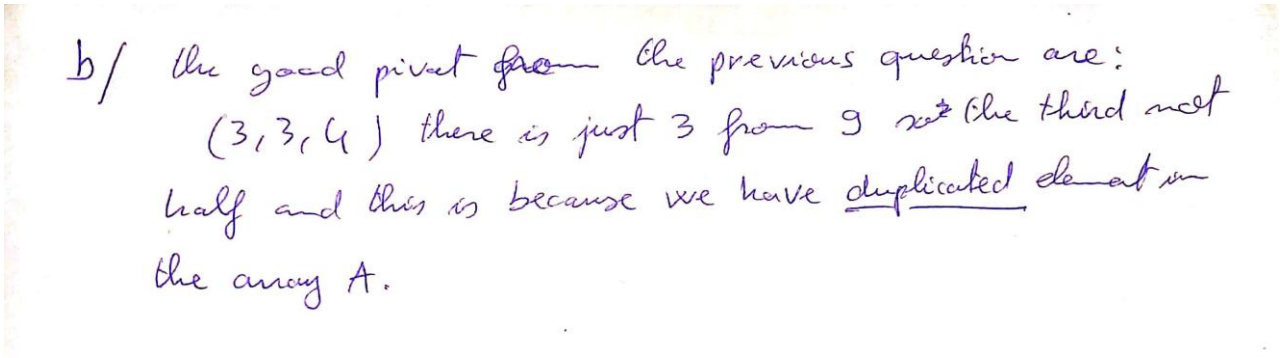


→ bothe of the number of elements in the right and left are <6 so 3 is a good pivot.





→ bothe of the sets of elements contain <6 element so 4 is a good pivot



4/

```
public static boolean equalIndex(int[] a, int i, int j) {

		if(i==a[i] || j==a[j]) {
			System.out.println("i= "+i +"     " + "j= "+j );
			return true;
		}
		if(i!=a[i] && j!=a[j] && (i==j || j==i+1)) return false;

		if(i<a[i] && j>a[j] && (i+j)/2<=a[(i+j)/2]) return equalIndex(a,
(i+j)/2, j);
		else if(i<a[i] && j>a[j] && (i+j)/2>=a[(i+j)/2]) return equalIndex(a, i,
(i+j)/2);
		else if(i>a[i] && j<a[j] && (i+j)/2>=a[(i+j)/2]) return equalIndex(a,
(i+j)/2, j);
		else if(i>a[i] && j<a[j] && (i+j)/2<=a[(i+j)/2]) return equalIndex(a, i,
(i+j)/2);

		return false;
	}
```

This Algorithm has running time of O(log(n)) which is o(n) because each self-call the algorithm divides the array by half: $a/2^0$, $a/2^1$, $a/2^2$ … $a/2^n$, n is the number of self-calls and a is the length of the array. In the worst case:

$a/2^n=1$ => $a=2^n$ => $n=\log(a)/\log(2)$ => $O(\log(n))$

5/

```
static void SubsetSum(int arr[], int n, List<Integer> l, int sum) {

		if (sum == 0) {
			for (int i = 0; i < l.size(); i++)
			System.out.print(l.get(i) + " ");
			System.out.println();
			return;
		}

		if (n == 0)
			return;


		SubsetSum(arr, n - 1, l, sum);
		List<Integer> v1 = new ArrayList<Integer>(l);
		v1.add(arr[n - 1]);
		SubsetSum(arr, n - 1, v1, sum - arr[n - 1]);
	}
```