1. BinarySearch

**public int** findFirstTarget(**int**[] *nums*, **int** *target*) {  
 **if** (*nums* == **null** || *nums*.length == **0**) { **return** -**1**; }  
  
 **int** start = **0**, end = *nums*.length - **1**;  
// break when adjacent  
// break when start = 1, end = 2  
 **while** (start + **1** < end) {  
// int mid = (start + end) / 2 Exceed  
 **int** mid = start + (end - start) / **2**;  
  
 **if** (*nums*[mid] == *target*) {  
 end = mid;  
 } **else if** (*nums*[mid] < *target*) {  
 start = mid;  
 } **else** {  
 // same as end = mid - 1  
 end = mid;  
 }  
 }   
 // double check  
 **if** (*nums*[start] == *target*) {  
 **return** start;  
 }  
 **if** (*nums*[end] == *target*) {  
 **return** end;  
 }  
 **return** -**1**;  
}

1. Find K closest elements

**public List**<**Integer**> findClosestElements2(**int**[] *arr*, **int** *k*, **int** *target*) {  
 **int** start = **0**, end = *arr*.length - *k*;  
  
 **while** (start < end) {  
 **int** mid = start + (end - start) / **2**;  
 **int** curr = *arr*[mid],

kcurr = *arr*[mid + *k*];  
  
**if** (**2** \* *target* > kcurr + curr) {  
 start = mid + **1**;  
 } **else** {  
 end = mid;  
 }  
 }  
**List**<**Integer**> results = **new** ArrayList<**Integer**>();  
**for** (**int** i = start;i < start + *k*;i++) {  
 results.add(*arr*[i]);  
 }  
 **return** results;  
}

1. Copy Books

**public int** copyBooks(**int**[] *pages*, **int** *people*) {  
**if** (*pages* == **null** || *pages*.length == **0**) { **return 0**; }  
  
**int** totalTime = **0**, maxTimePerPage = **0**;  
**for**(**int** i = **0**; i < *pages*.length; i++) {  
 totalTime += *pages*[i];  
 **if** (*pages*[i] > maxTimePerPage) {  
 maxTimePerPage = *pages*[i];  
 }  
 }  
  
**int** start = maxTimePerPage,

end = totalTime;  
 **while** (start + **1** < end) {  
 **int** mid = start + (end - start) / **2**;  
  
**if**(countPeople(mid, *pages*) <= *people*) {  
 end = mid;  
 } **else** {  
 start = mid;  
 }  
 }  
 // as small as possible thus start first then end  
**if**(countPeople(start,*pages*)<=*people*) {  
 **return** start;  
 }  
**if**(countPeople(end, *pages*) <= *people*) {  
 **return** end;  
 }  
 **return 0**;  
}  
  
**private int** countPeople(**int** *maxTime*, **int**[] *arr*) {  
 **int** headCount = **1**;  
 **int** sumTime = *arr*[**0**];  
  
 **for** (**int** i = **1**; i < *arr*.length; i++) {  
 **if** (sumTime + *arr*[i] > *maxTime*) {  
 headCount ++;  
 sumTime = **0**;  
 }  
 sumTime += *arr*[i];  
 }  
 **return** headCount;  
}

1. ReverseLinkedList

// Iterator  
**public ListNode** reverseList(**ListNode** head) {  
**if** (*head* == **null** || *head*.next == **null**) { **return** *head*; }  
 **ListNode** prev = **null**;  
 **while** (*head* != **null**) {  
 **ListNode** temp = *head*.next;

// keep next node  
 *head*.next = prev;

// curr points to (old)prev  
 prev = *head*; // renew prev to curr  
 head = temp;

// renew curr to (old)curr.next  
 } // curr == null, prev = (old) curr  
 **return** prev;  
}  
// Recursion  
**public ListNode** reverseList2(**ListNode** *head*) {  
**if** (*head* == **null** || *head*.next == **null**) { **return** *head*; }  
 **return** reverseListRec(*head*, **null**);  
}

**private ListNode** reverseListRec(**ListNode** *curr*, **ListNode** *newNext*) {  
**if** (*curr* == **null**) {  
 **return** *newNext*;

// at end, return last valid node  
 }  
 **ListNode** oldNext = *curr*.next;  
 *curr*.next = *newNext*;  
 **return** reverseListRec(oldNext, *curr*);  
}

1. Add Two Numbers

/\*\* Optimal Passed in 39 ms   
 \* Time Complexity O(max(m,n))  
 \* Space Complexity O(max(m,n))  
 \*/  
**private ListNode** addTwoNumbersX(**ListNode** l1, **ListNode** l2) {  
 **int** thisRound = **0**, stageSum = **0**;  
// Use tail to hold root, a better way than mine  
// Base root is useless, but don't need to create root inside of the loop  
**ListNode** root = **new** ListNode(**0**),

tail = root;  
  
 **while** (!(*l1* == **null** && *l2* == **null** && thisRound == **0**)) {  
 **int** num1 = (*l1* != **null**) ? *l1*.val : **0**;  
 **int** num2 = (*l2* != **null**) ? *l2*.val : **0**;  
   
//stage sum will never beyond 27(9+9+9)  
 stageSum = num1 + num2 + thisRound;  
 // thisRound must be 0 - 9  
 thisRound = stageSum / **10**;  
  
// A bit slower: sum>=10 ? sum-10 : sum  
 tail.next = **new** ListNode(stageSum % **10**);  
 tail = tail.next;  
  
 **if** (*l1* != **null**)  
 l1 = l1.next;  
 **if** (*l2* != **null**)  
 l2 = l2.next;  
 }  
 **return** root.next;  
}

1. Search 2D Matrix II

**public boolean** searchMatrix(**int**[][] *matrix*, **int** *target*) {  
 **int** rows = *matrix*.length;  
 **if** (*matrix* == **null** || rows == **0** || *matrix*[**0**].length == **0**) {  
 **return false**; }  
  
 **int** cols = *matrix*[**0**].length;  
 **int** currRow = **0**, currCol = cols - **1**;  
**while**(currRow < rows && currCol >= **0**) {  
**int** currNum = *matrix*[currRow][currCol];  
  
 **if** (*target* == currNum) {  
 **return true**;  
 }  
 **if** (*target* < currNum) {  
 currCol --;  
 } **else** { // target > currNum  
 currRow ++;  
 }  
 }  
 **return false**;  
}

1. Search 2D Matirx I

**public boolean** searchMatrix2(**int**[][] *matrix*, **int** *target*) {  
 **int** rows = *matrix*.length;  
 **if** (*matrix* == **null** || rows == **0** || *matrix*[**0**].length == **0**) {  
 **return false**; }  
 **int** rowTop = **0**, rowBottom = rows - **1**, cols = *matrix*[**0**].length;  
  
 **while** (rowTop <= rowBottom) {  
**int** mid=rowTop+(rowBottom-rowTop)/**2**;  
 **int** currHead = *matrix*[mid][**0**], currTail = *matrix*[mid][cols - **1**];  
  
// wrote target >= currTail before.  
 **if** (currHead <= *target* && *target* <= currTail) {  
 // This can make sure target is in this line or not exist  
**return** searchArray(*matrix*[mid],*target*);  
 }  
 **if** (*target* < currHead) {  
 rowBottom = mid - **1**;  
 } **else** {  
 rowTop = mid + **1**;  
 }  
 }  
 // if it's here, then no row match for target.  
 **return false**;  
}

1. Rotate String

**public void** rotateString(**char**[] *str*, **int** offset) {  
 **int** N = *str*.length;  
 **if** (*str* == **null** || N <= **1**) {  
 **return**;  
 }  
offset = offset % N;  
 reverse(*str*, **0**, N - offset - **1**);  
 reverse(*str*, N - offset, N - **1**);  
 reverse(*str*, **0**, N - **1**);  
}  
  
**private void** reverse(**char**[] *cArr*, **int** start, **int** end) {  
**for** (; *start* < *end*; start++, end--) {  
 **char** temp = *cArr*[start];  
 *cArr*[start] = *cArr*[end];  
 *cArr*[end] = temp;  
 }  
}

1. Binary Pre Order Traversal

// Rec

**public List**<**Integer**> preorderTraversal(**TreeNode** *root*) {  
 **List**<**Integer**> result = **new** ArrayList<>();  
 **if** (*root* == **null**) {  
 **return** result;  
 }  
 preorderRec(*root*, result);  
 **return** result;  
}  
/\*\*  
 \* put preorder nodes rooted in root into result  
 \*/  
**private void** preorderRec(**TreeNode** *curr*, **List**<**Integer**> *result*) {  
 **if** (*curr* == **null**) {  
 **return**;  
 }  
 *result*.add(*curr*.val);  
 preorderRec(*curr*.left, *result*);  
 preorderRec(*curr*.right, *result*);  
}  
  
/\*\*  
 \* Iterate, Time O(n), Space O(n)  
 \*/  
**public List**<**Integer**> preorderTraversal2(**TreeNode** *root*) {  
 **List**<**Integer**> result = **new** ArrayList<>();  
 **if** (*root* == **null**) {  
 **return** result;  
 }  
 **Deque**<**TreeNode**> stack = **new** ArrayDeque<>();  
 stack.push(*root*);  
  
 **while** (!stack.isEmpty()) {  
 **TreeNode** node = stack.pop();  
// if this is left, then if there exist next left  
// we can make sure that's the next one gona be popped  
 // and added to the tail  
 result.add(node.val);  
  
 **if** (node.right != **null**) { // when pop, this goes after left  
 stack.push(node.right);  
 }  
 **if** (node.left != **null**) { // when pop, this goes first  
 stack.push(node.left);  
 }  
 }  
  
 **return** result;  
}

1. Inorder Traversal

/\*\*  
 \* Iterator (dfs), Time O(n), Space O(n)  
 \*/  
**public List**<**Integer**> inorderTraversal2(**TreeNode** *root*) {  
 **List**<**Integer**> result = **new** ArrayList<>();  
 **if** (*root* == **null**) {  
 **return** result;  
 }  
 **TreeNode** curr = *root*; // not necessary  
 **Deque**<**TreeNode**> stack = **new** ArrayDeque<>();  
  
 **while** (**true**) {  
 // inorder: left, root, right  
 // add all left nodes into stack  
 **while** (curr != **null**) {  
 stack.push(curr);  
 curr = curr.left;  
 } // curr == null  
 // if stack is empty, then finished  
 **if** (stack.isEmpty()) {  
 **break**;  
 }  
 // all left nodes in stack  
 // if right node empty, pop, else add right  
 curr = stack.pop();  
 // add left and root  
 result.add(curr.val);  
 // if right is null, then next round pop stack  
 // else next round add right.left first ...  
 curr = curr.right;  
 }  
 // stack is empty  
 **return** result;  
}

1. PostOrder Traversal

/\*\*  
 \* Iterating, dfs, Time O(n), Space O(n)  
 \*/  
**public List**<**Integer**> postorderTraversal2(**TreeNode** *curr*) {  
 **LinkedList**<**Integer**> result = **new** LinkedList<>(); // addFirst O(1)  
 **if** (*curr* == **null**) {  
 **return** result;  
 }  
 **Deque**<**TreeNode**> stack = **new** ArrayDeque<>();  
 stack.push(*curr*);  
  
 **while** (!stack.isEmpty()) {  
 **TreeNode** node = stack.pop();  
 result.addFirst(node.val); // addFirst is key  
  
 **if** (node.left != **null**) { // when pop, this goes later but add before right  
 stack.push(node.left);  
 }  
 **if** (node.right != **null**) { // when pop, this goes first but add after left  
 stack.push(node.right);  
 }  
 }  
 **return** result;  
}

1. SubTree with maximum avg

**public TreeNode** findSubtree2(**TreeNode** *root*) {  
 **ResultType** result = dcHelper(*root*);  
 **return** result.maxNode;  
}  
  
// Think about that we need to get the result  
// 1. number of nodes in subtree rooted in node  
// 2. current average  
// 3. subtree node with maximum average  
**private class ResultType** {  
 **TreeNode** maxNode;  
 **double** maxAvg;  
 **int** sum, size;  
  
 **public ResultType**(**TreeNode** *maxNode*, **double** *maxAvg*, **int** *sum*, **int** *size*) {  
 **this**.maxNode = *maxNode*;  
 **this**.maxAvg = *maxAvg*;  
 **this**.sum = *sum*;  
 **this**.size = *size*;  
 }  
}  
  
// Think about that we need in this helper  
// We want to find the things defined in ResultType  
**private ResultType** dcHelper(**TreeNode** *node*) {  
 **if** (*node* == **null**) { // These three values are important! Leaf Node  
 **return new** ResultType(*node*, -**Double**.***MAX\_VALUE***, **0**, **0**);  
 } // Double.MIN\_VALUE is another thing  
 // Divide  
 **ResultType** left = dcHelper(*node*.left);  
 **ResultType** right = dcHelper(*node*.right);  
  
 **int** currSize = left.size + right.size + **1**;  
 **int** currSum = left.sum + right.sum + *node*.val;  
 **double** currAvg = (**double**) currSum / currSize; // cast is a must  
 **ResultType** currResult = **new** ResultType(*node*, currAvg, currSum, currSize);  
  
 // renew ResultType, Note must call currResult.maxAvg and not currAvg!!!  
 **if** (left.maxAvg > currResult.maxAvg) {  
 currResult.maxNode = left.maxNode;  
 currResult.maxAvg = left.maxAvg;  
 }  
 **if** (right.maxAvg > currResult.maxAvg) {  
 currResult.maxNode = right.maxNode;  
 currResult.maxAvg = right.maxAvg;  
 }  
 // else currAvg >= left/right maxAvg  
 **return** currResult;  
}

1. Binary Tree Paths

**public List**<**String**> binaryTreePaths(**TreeNode** *root*) {  
 **List**<**String**> result = **new** ArrayList<>();  
 **if** (*root* == **null**) {  
 **return** result;  
 }  
 pathHelper(*root*, **String**.**valueOf**(*root*.val), result);  
 **return** result;  
}  
  
**private void** pathHelper(**TreeNode** *root*, **String** *path*, **List**<**String**> *result*) {  
 **if** (*root* == **null**) { // avoid NPE  
 **return**;  
 }  
 // Make sure this is the end  
 **if** (*root*.left == **null** && *root*.right == **null**) {  
 *result*.add(*path*);  
 **return**;  
 }  
 **if** (*root*.left != **null**) {  
 pathHelper(*root*.left, *path* + **"->"** + **String**.**valueOf**(*root*.left.val), *result*);  
 }  
 **if** (*root*.right != **null**) {  
 pathHelper(*root*.right, *path* + **"->"** + **String**.**valueOf**(*root*.right.val), *result*);  
 }  
}