Table of Contents

[FindMinSortedArray 2](#_Toc510510192)

[FindMinSortedRotatedArray2 2](#_Toc510510193)

[SearchSortedRotatedArray 3](#_Toc510510194)

[SearchSortedRotatedArray2 3](#_Toc510510195)

[SquareRoot 4](#_Toc510510196)

[First Bad Version 4](#_Toc510510197)

[Buy Sell Stocks – One Transaction 4](#_Toc510510198)

[Buy Sell Stocks – Two Transaction 5](#_Toc510510199)

[Buy Sell Stocks – Infinite Transactions 5](#_Toc510510200)

[Buy Sell Stocks – K Transactions 6](#_Toc510510201)

[Buy Sell Stocks – Infinite Transactions + Cool Down 6](#_Toc510510202)

[Buy Sell Stocks – Infinite Transactions + Transaction Fee 7](#_Toc510510203)

[AddTwoNumbers (LinkedList) 7](#_Toc510510204)

[Binary Addition (2 Strings) 8](#_Toc510510205)

[Average of Binary Tree Levels 8](#_Toc510510206)

[HeightBalancedBinaryTree 9](#_Toc510510207)

[BinaryTreeLevelOrderTraversal 9](#_Toc510510208)

[Root-to-leaf paths 9](#_Toc510510209)

[CheckSubTree 10](#_Toc510510210)

[ClimbingStairs 11](#_Toc510510211)

# FindMinSortedArray

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element. You may assume no duplicate exists in the array.

**Solution:**

middle = left + (right - left) / 2;

if middle is greater than right, more left +1. Stay there and move the left until it hits left < right and exit.

# FindMinSortedRotatedArray2

"Find Minimum in Rotated Sorted Array": What if duplicates are allowed?

**Solution:**

Go until left <= right, with extra conditions while arr[left] == arr[right] and left <= right, left ++

If arr[left] <=arr[right], break out. Otherwise binary search as before

# SearchSortedRotatedArray

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

**Solution:**

Go until left <= right, if key == middle, return.

if left <= middle,

if left <= key < middle, then right = middle - 1

else left = middle + 1

else

if middle <= key < right, then left = middle + 1

else right = middle - 1

# SearchSortedRotatedArray2

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed?

**Solution:**

Extra condition to go one by one if equals. As in find min index

Go until left <= right, if key == middle, return.

if left < middle,

if left <= key < middle, then right = middle - 1

else left = middle + 1

else if left > middle

if middle <= key < right, then left = middle + 1

else right = middle – 1

else left == middle,

just go left ++

# SquareRoot

Implement int sqrt(int x). Compute and return the square root of x. x is guaranteed to be a non-negative integer.

**Solution:**

Middle = left + (Right – left)/2. Square = (double) middle \* (double) middle.

square > target, right = middle -1

square < target, left = middle + 1

# First Bad Version

n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

**Solution:**

Middle = left + (right – left) /2

If (middle is not bad)

left = middle +1;

Else

Right = middle

# Buy Sell Stocks – One Transaction

Input: [7, 1, 5, 3, 6, 4]

Output: 5

max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

**Solution:**

Maintain three variables for max profit at the end of each day,

0TransactionHold0 = 0

1TransactionHold0 = 0

1TransactionHold1 = Integer.Min\_value.

Go through each price, and calculate the vals

1TransactionHold0 = max (1TransactionHold0 (hold), 1TransacctionHold1 + price (sell))

1TransacctionHold1 = max (1TransacctionHold1 (hold), 0TransactionHold0 - price (buy))

Return 1TransactionHold0

**NOTE:** Always start from higher transactions to lower transactions hold0 – to use previous values next time

# Buy Sell Stocks – Two Transaction

At most two transactions

**Solution:**

Maintain five variables for max profit at the end of each day,

0TransactionHold0 = 0

1TransactionHold0 = 0

1TransactionHold1 = Integer.Min\_value

2TransactionHold0 = 0

2TransactionHold1 = Integer.Min\_value

Go through each price, and calculate the vals

2TransactionHold0 = max (2TransactionHold0 (hold), 2TransactionHold1 – price(sell))

2TransactionHold1 = max (2TransactionHold1 (hold), 1TransactionHold0 + price (sell))

1TransactionHold0 = max (1TransactionHold0 (hold), 1TransacctionHold1 + price (sell))

1TransacctionHold1 = max (1TransacctionHold1 (hold), 0TransactionHold0 - price (buy))

Return 2TransactionHold0

# Buy Sell Stocks – Infinite Transactions

As many transactions as possible.

**Solution:**

If the hold1 takes the max of Prev hold0 – price (buy), then it accumulates the transactions.

prevTransactionHold0 = 0

TransactionHold0 = 0

TransactionHold1 = Integer.Min\_value.

Go through each price, and calculate the vals

prevTransactionHold0 = TransactionHold0;

TransactionHold0 = max (TransactionHold0 (hold), 1TransacctionHold1 + price (sell))

TransactionHold1 = max (TransactionHold1 (hold), prevTransactionHold0 - price (buy))

Return 1TransactionHold0

# Buy Sell Stocks – K Transactions

Specified number of transactions allowed.

**Solution:**

After, n/2 you cannot do more transactions as its always buy at one day and sell at one day. So, for if k > n/2, do infinite transactions, for k < n/2 do array’s

So, maintain two arrays for holding the values.

Hold0 array filled with 0 with k + 1 size.

Hold1 array filled with -1 with k + 1 size. To start with base condition

Go through each price, and calculate the vals

For each price, j = k, j until 0

Hold0[j] = max (hold0[j], hold1[j] + price) sell

Hold1[j] = max (hold1[j], hold0[j-1] - price) buy

# Buy Sell Stocks – Infinite Transactions + Cool Down

Wait one day before starting new transaction.

**Solution:**

If the hold1 takes the max of Prev hold0 – price (buy), then it accumulates the transactions.

prevTransactionHold0 = 0

TransactionHold0 = 0

TransactionHold1 = Integer.Min\_value.

Go through each price, and calculate the vals

prevPrevTransactionHold0 = prevTransactionHold0

prevTransactionHold0 = TransactionHold0;

TransactionHold0 = max (TransactionHold0 (hold), 1TransacctionHold1 + price (sell))

TransactionHold1 = max (TransactionHold1 (hold), prevPrevTransactionHold0 - price (buy))

Return TransactionHold0

# Buy Sell Stocks – Infinite Transactions + Transaction Fee

Pay fee for each time

**Solution:**

Reduce the fee from profit;

If the hold1 takes the max of Prev hold0 – price (buy), then it accumulates the transactions.

prevTransactionHold0 = 0

TransactionHold0 = 0

TransactionHold1 = Integer.Min\_value.

Go through each price, and calculate the vals

prevTransactionHold0 = TransactionHold0;

TransactionHold0 = max (TransactionHold0 (hold), 1TransacctionHold1 + price (sell))

TransactionHold1 = max (TransactionHold1 (hold), prevTransactionHold0 - price (buy) - fee)

Return 1TransactionHold0

**NOTE:** Take out while buying to avoid overflow…

# AddTwoNumbers (LinkedList)

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

**Solution:**

Create new list nodes to store result.Only one loop until either one is available, l1 or l2 or carry.

If l1 is null, val is zero. L2 is null, val is zero.

Sum = l1.val + l2.val + carry

currentVal = sum %10

carry = sum / 10

# Binary Addition (2 Strings)

a = "11"

b = "1"

Return "100"

**Solution:**

i = size of a – 1 (lsb)

j = size of b -1 (msb)

Only one loop until the end of a or b

Int sum = 0

If a is not done && a[i] == 1

Sum ++

If b is not done && b[i] == 1

Sum ++

Sum += carry

Carry = sum / 2;

Insert at beginning of result string (Char) (sum %2) + ‘0’

# Average of Binary Tree Levels

Return the average value of the nodes on each level in the form of an array.

Example 1:

Input:

3

/ \

9 20

/ \

15 7

Output: [3, 14.5, 11]

**Solution:**

Create a queue to hold the nodes in a level. Check until queue is empty – outer loop. Get all the elements at a time in the queue. 🡪 empty it by calculating the n. Loop through all the elements and find the average. At the same time, add the left and rights to the same queue for next level.

# HeightBalancedBinaryTree

Given a binary tree, determine if it is height-balanced.

**Solution:**

Condition: a node is balanced if left subtree and right subtree is balanced + number of nodes in right subtree – left subtree <= 1

Create a wrapper to hold isBalanced, height.

Report height from bottom with Max (left subtree height, right subtree height) + 1 (current level)

# BinaryTreeLevelOrderTraversal

For example:

Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its level order traversal as:

[

[3],

[9,20],

[15,7]

]

**Solution:**

Same as average of levels. Get the queue size at each iteration. Empty until the size and add the next levels into the same queue. Calculate result when iterating through the inner n elements of the queue

# Root-to-leaf paths

1

/ \

2 3

\

5

All root-to-leaf paths are:

["1->2->5", "1->3"]

**Solution:**

Use recursive helper,

Base condition root == null,

Then add root, to a list of current nodes.

Do (required operation) when left == null && right == null.

Creating the string of paths.

If (left != null) {

Do left,

Remove current nodes.size() – 1.

}

If (right != null) {

Do right,

Remove current nodes.size() – 1.

}

# CheckSubTree

3

/ \

4 5

/ \

1 2

/

0

Given tree t:

4

/ \

1 2

Return false.

**Solution:**

Subtree:

If (t1.val is same t2.val && equals tree (t1, t2)

Return true;

Else

check subtree (t1.left, t2) OR check subtree(t1.right, t2)

Equals Tree:

Base condition: true

t1 == null && t2 == null

Base condition: False

t1 == null && t2 != null, t1 != null && t2 == null

T1.val == t2. Val && Equals Tree(t1.left, t2.left) && Equals Tree(t1.right, t2.right)

# ClimbingStairs

Input: 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

**Solution:**

Fibonacci sequence

0 step - 1

1 step – 1

2 step – 2

Loop through from 3 steps to k steps

3 steps = 2 step + 1 step

1step = 2 step;

2step = 3 step;