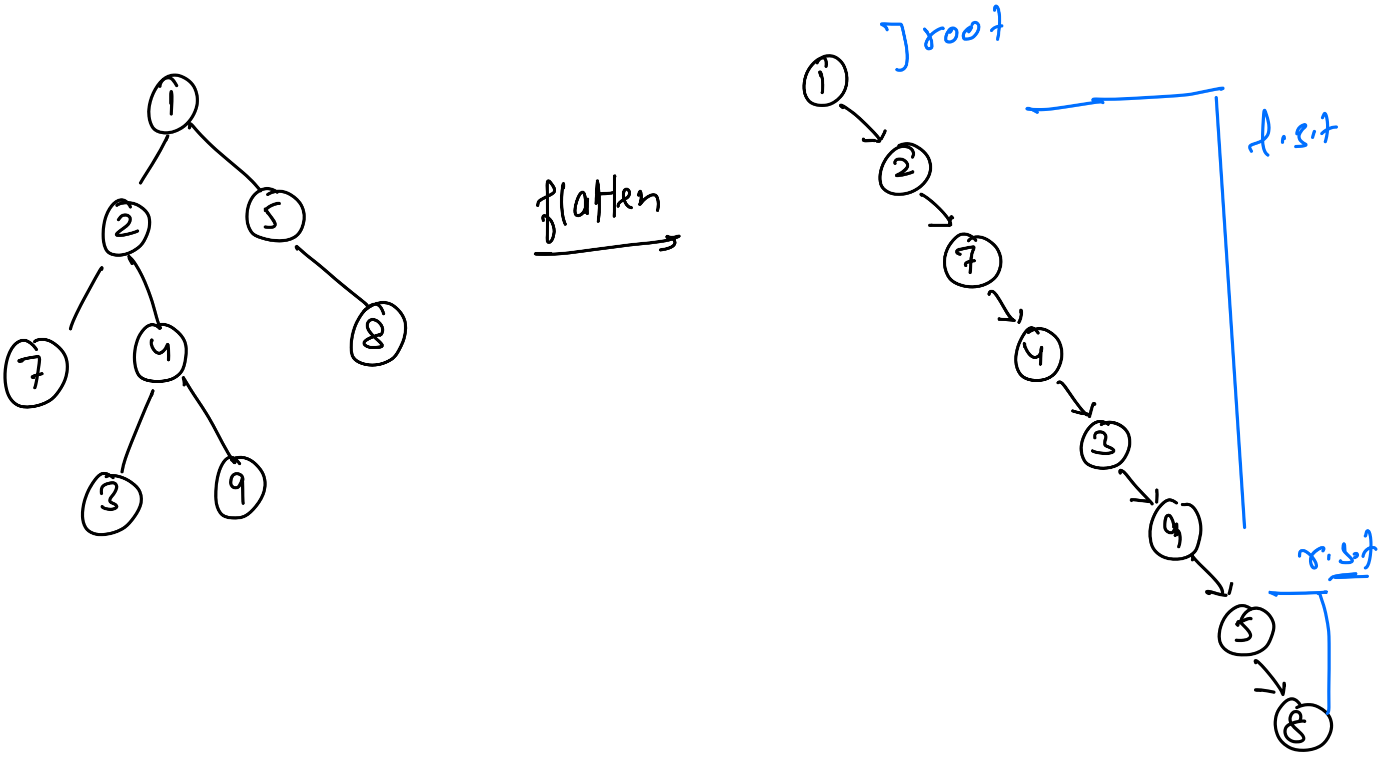
✓ → Flatten Binary Tree to Linked List

✓ → Insert , Delete , Get Random → O(1)

✓ → Best time to buy & sell stock.

✓ → Partition to K Equal Subsets.

① Flatten Binary Tree to Linked-list.

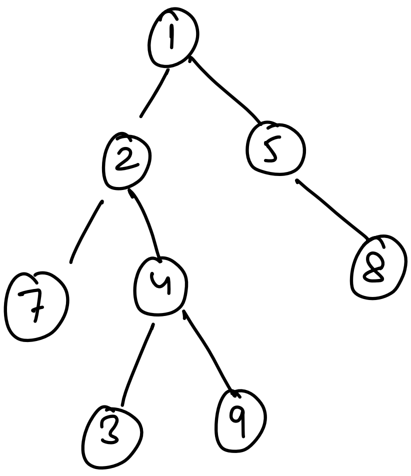


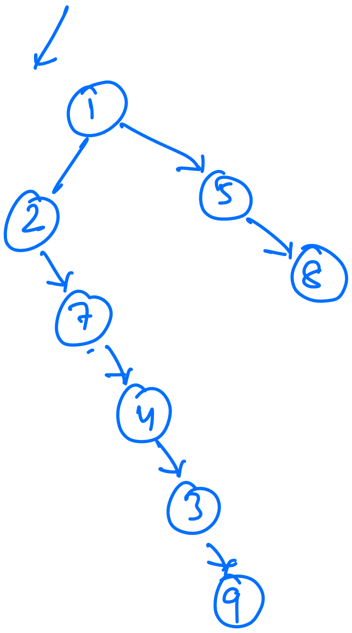
ans ⇒ 1 → 2 → 7 → 4 → 3 → 9 → 5 → 8

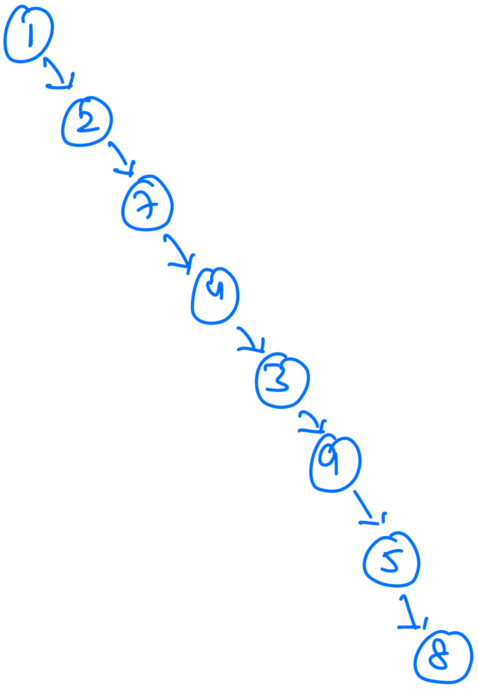
A1 : Apply pre-order traversal & for every node make a new node and include in your ans.

T.C → O(N)
S.C → O(N)

idea.2







```
class Pair {
        Node head;
        Node tail;
}
```

code ·→

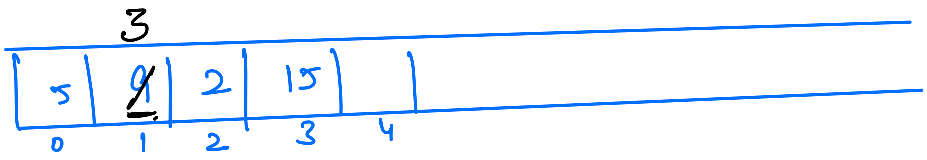
```
Pair flatten( Node root){
    if(root == NULL) { return new Pair(null,null); }

        Pair lp = flatten(root.left);
        Pair rp = flatten(root.right);

        if(lp.head == NULL && rp.head == NULL){
                return new Pair(root, root);
        }
        else if (lp.head == NULL){
                return new Pair(root, rp.tail);
        }
        else if ( rp.head == NULL){
                root.left = NULL;
                root.right = lp.head ;
                return new Pair( root, lp.tail);
        }
        else{
                root.left = NULL;
                root.right = lp.head;
                lp.tail.right = rp.head;
                return new Pair (root, rp.tail);
        }
}
```

T.C → O(N)
S.C → O(Ht.)

Q2:)

Implement the RandomizedSet class:

RandomizedSet() Initializes the RandomizedSet object.
bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the same probability of being returned.
You must implement the functions of the class such that each function works in average O(1) time complexity.



element	idx
5 →	0
~~9 →~~	~~1~~
2 →	2
15 →	3
3 →	~~4~~ 1

insert(5) ✓
insert(9) ✓
insert(2) ✓
insert(15) ✓
remove(8) ← false.
insert(2) ← false
insert(3) ✓
getRandom()
remove(9)

```
Random random = new Random();
al.get(random.nextInt(list.size()));
```

code.→

```
class    Randomizedset {

    HashMap < Integer, Integer > map;

    ArrayList <Integer> list;

    public  RandomizedSet ( ) {
            map = new HashMap <>();
            list = new ArrayList <>();
    }

    boolean insert ( int val) {
        if (map.containsKey (val) == true) {
                return false;
        }
                                                  → O(1)
        al.addLast (val);
        map.put( val, al.size() - 1);
        return true;
    }

    int getRandom () {
        Random random = new Random();             → O(1)
        return al.get( random.nextInt( list.size()));
    }
boolean remove( int val){
        if(map.containsKey(val) == false){
              return false;
        }
         int idx = map.get(val);
          swap( al[idx] with al[al.size() -1]);
           map.put( al[idx], idx);
          al.remove( al.size() -1);                 => O(1)
            map.remove( val);
            return true;
}
}
```

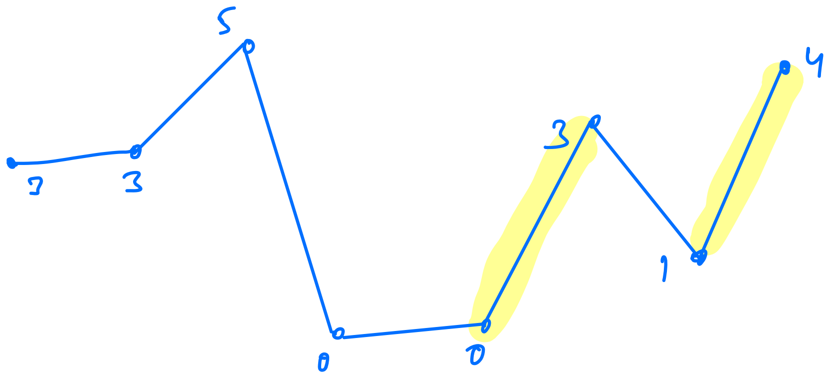
Q.1

You are given an array A, where the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit you can achieve by completing at most 2 transactions. Note that you cannot engage in multiple transactions at the same time, meaning you must sell the stock before buying again.

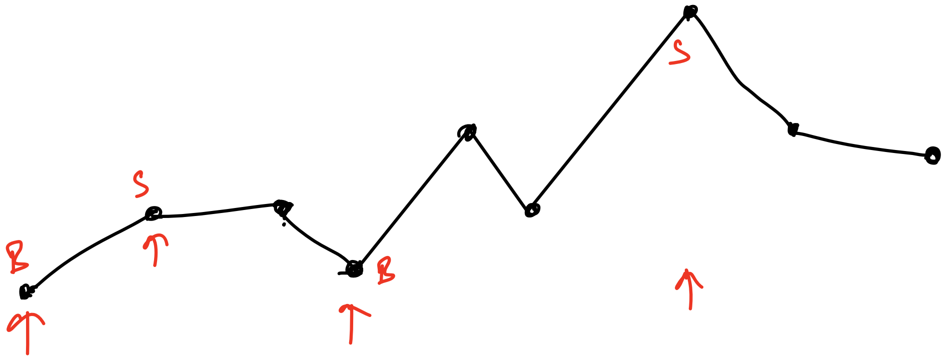
1 transaction (Buy → Sell] [Buy → Sell]

arr[] → [3, 3, 5, 0, 0, 3, 1, 4]
0 1 2 3 4 5 6 7

ans = 6.



arr[] → [3 6 6 2 6 4 9 3 1]



B.f. → Consider all the quadruplets   T.C → $O(n^4)$

idea.2. →

arr[] →	[3,	3,	5,	0,	0,	3,	1,	4]
	0	1	2	3	4	5	6	7
b1 →	-3	-3	-3	0	0	0	0	0
s1 →	0	0	2.	2	2	3	3	4
		0, -3+3						

for at max one transaction ⇒

```
firstbuy = -arr[0],  firstsell → 0

for( i=1; i<N; i++) {
    b1 = firstbuy;
    s1 = firstsell;

    firstbuy  = max( b1, -arr[i]);
    firstsell = max( s1, arr[i] + b1);
}
return firstsell;
```

for at max 2 transactions →

```
firstbuy = -arr[0], firstsell -> 0 , secondbuy = -∞ , secondsell -> 0

for( i=1; i < N; i++) {
        b1 = firstbuy; s1 = firstsell;
        b2 = secondbuy; s2 = secondsell;

        firstbuy = Max( b1, -arr(i]);
        firstsell = max( s1, arr[i] + b1);
        secondbuy = Max( b2, s1 - arr[i]);
        secondsell = Max( s2, b2 + arr(i]);

}

return Max( firstsell, secondsell);
```

T.C -> O(N)

S.C -> O(1)

4

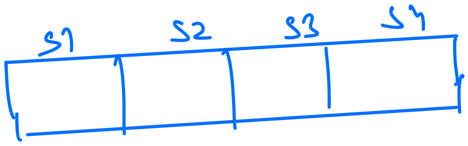
Given an integer array nums and an integer k, return true if it is possible to divide this array into k non-empty subsets whose sums are all equal.

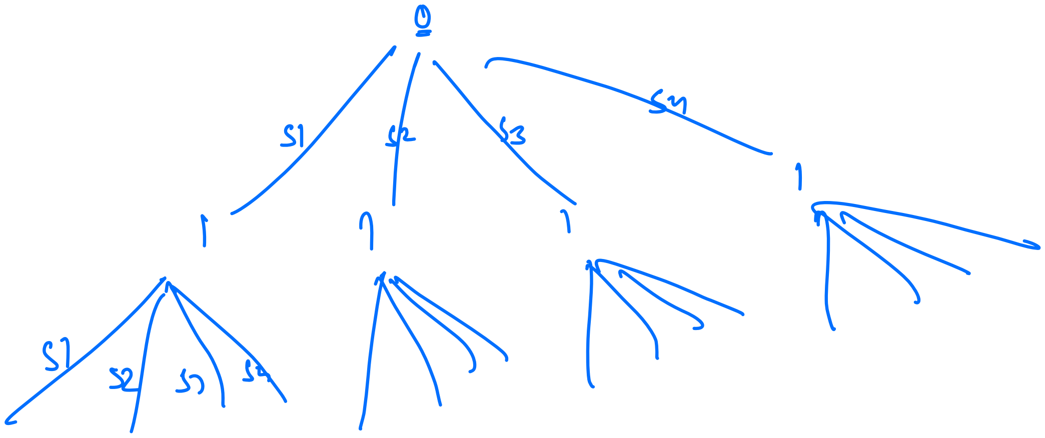
nums[] → [4, 3, 2, 3, 5, 2, 1] , k=4

[4,1] [5] [3,2] [3,2]   ans → true.

→ sum = 20 is divisible by k=4

Idea-1.





T.C → $O(k^N)$

idea-2.

Given an integer array nums and an integer k, return true if it is possible to divide this array into k non-empty subsets whose sums are all equal.

k=4

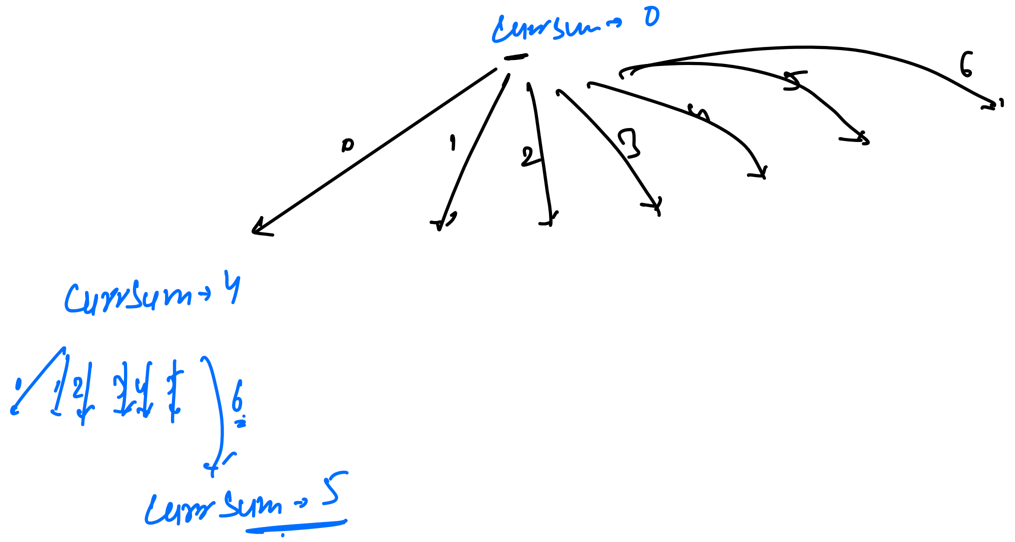
nums[] → [4, 3, 2, 3, 5, 2, 1]
0 1 2 3 4 5 6

vis[] → [~~f~~ T, f, f, f, f, f, ~~f~~ T]

sum of ss ⇒ $\frac{totalSum}{k}$

→ $\frac{20}{4} = \boxed{5}$

canPartition(nums[], K, 0, 0, totalsum/K, visited[]);



code.→

```
sum=0;
for( int val : nums[] ){
        sum += val;
}
if( sum % K != 0){ return false }
boolean visited [N];        ∀i, visited [i] = false;
return   canPartition( nums[], K, 0, 0, sum/K , visited[]);
                                  ↑   ↖
                        Start-index   currsum
```

```
boolean  canPartition( nums[], K, idx, currsum, targetSum, visited[] ){
  if(K==0){ return  true }
  if( currSum == targetSum){
          return  canPartition(nums[], K-1, 0, 0, targetSum visited);
  }
  for( i= idx ;  i < N ; i++){
        if( visited[i] == false  && currSum + nums[i] <= targetSum){
              visited[i] = true;
              if( canPartition(nums, K, i+1, currSum+ nums[i], targetSum,
                                                              visited[] )){
                     return  true;
              }
```

```
                visited(i) = false;
            }
    }
    return false;
}
```

T.C → $O(2^{n} \times K)$

S.C → [?]