VRIJE
UNIVERSITEIT
BRUSSEL

ML Algorithm Interview Preparation 2020

INTERVIEW PREPARATION

LiangLiang ZHENG

Academic Year: 2019-2020

Promotor:
Advisor:
Set a faculty using \faculty{*Engineering Sciences*}

Contents

Chapter 1

Machine Learning and Deep learning (FAQ)

Chapter 2

Math

Chapter 3

Algorithm

3.1 General Algorithmic Knowledge

3.1.1 alphabet changing

To know if a character is a alphabet, 'a'.isapha()

3.1.2 check valid parentheses

Check if a sequence of parentheses are valid, count the left paren first and if a right paren occurs and the left paren is 0, which means before this right paren there must is no left paren that could be corresponded to it. The code below is the code, which will be used in #301.

```
def valid(s):
        l = 0
        r = 0
        for i in s:
            if i == '(':
                l += 1
            if i == ')' and l == 0:
                r += 1
            elif i == ')':
                l -= 1
        return l == 0 and r == 0
```

3.1.3 Time Complexity and Space Complexity

3.1.4 Bit operations, Mod and Pow

Alphabet case conversion XOR operation with ' ', for example, converting 'd' to 'D' or do it reversely, simply do $chr(ord('d')^ord(''))$

In #952, mod is always used in circle like number like circular list. A circle like number 0-9 where 0 - 1 = 9 and 9 + 1 = 0, the easiest way to know what number to mod could produce 0, in this case only 10 could do that.

3.1.5 count frequency

```
self.count = collections.Counter()
self.inorder(root)
freq = max(self.count.values())
res = []
for item, c in self.count.items():
    if c == freq:
        res.append(item)
return res
```

3.2 Basic Data Structure

3.2.1 Coding Template

3.2.1.1 Binary Search

```
def left_bound(nums, target):
    l = 0
    r = len(nums)
    while l < r:
        mid = l + (r - l) // 2
        if nums[mid] >= target:
            r = mid
        elif nums[mid] < target:
            l = mid + 1

    return l

def right_bound(nums, target):
    l = 0
    r = len(nums)
    while l < r:
        mid = l + (r - l) // 2
        if nums[mid] <= target:
            l = mid + 1
        else:
            r = mid
    return l - 1
```

3.2.1.2 Tree

inorder traversal: recursion
inorder traversal: iterative
pre-order and postorder: recusion
pre-order and postorder: iterative
level-order iterative

dfs build graph: template for turning a tree to a non-directed graph

```
graph = collections.defaultdict(list)
def dfs(parent, child):
    if parent and child:
        graph[parent.val].append(child.val)
        graph[child.val].append(parent.val)

    if child.left: dfs(child, child.left)
    if child.right: dfs(child, child.right)
dfs(None, root)
```

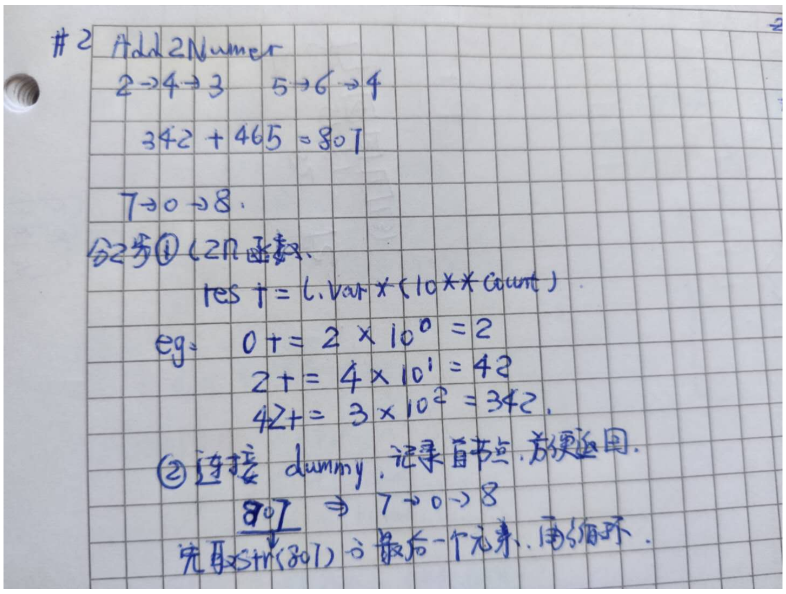
3.2.2 Array

3.2.3 Linked List

3.2.3.1 commonly used tricks

- dummy as previous node of head (#2,#445, #24)
- if not head or not head.next (#206, #24)
- fast-slow pointers(#141, #142, #876)

Example: In **Leetcode#2**, after transfering list to number using the function **l2n**, the result has been transferred to a string, in this way every element inside can be gotten simply by using index number, and in the end the list can be built in a iterative way. A dummy is initialized as head and will be used as return pointer because in iteration head is already changed. For loop is backward, From len - 2 (len - 1 should be assigned to head before) to the end 0.



#2 Add2Numer

$2 \to 4 \to 3 \quad 5 \to 6 \to 4$

$342 + 465 = 807$

$7 \to 0 \to 8.$

分2步 ① l2n函数.

res += l.var × (10 ** count).

eg: $0 += 2 \times 10^0 = 2$

$2 += 4 \times 10^1 = 42$

$42 += 3 \times 10^2 = 342.$

② 连接 dummy，记录首节点，方便返回.

$807 \Rightarrow 7 \to 0 \to 8$

先取str(807)的最后一个元素，再循环.

Figure 3.1: two sum

```
class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        def l2n(l):
            res = 0
            count = 0
            while l:
                res += l.val * (10 ** count)
                l = l.next
                count = count + 1

            return res

        str_add = str(l2n(l1) + l2n(l2))
        head = ListNode(str_add[len(str_add) - 1])

        dummy = head

        for i in range(len(str_add) - 2, -1, -1):
            node = ListNode(str_add[i])
            head.next = node
            head = head.next

        return dummy
```

Leetcode#445, add two number II, the only difference is that the function **l2n** is difference. $res = res * 10 + l.val$, and in for loop in index i is forward, from 1 (0 should be assigned to head before) to the end len - 1.

3.2.3.2 Reverse Linked List

Leetcode#206 is to reverse linked list, there are iterative way and recursive way to reverse the linked list. The first one below is **iterative** way, the important message is that whenver you want to assign a pointer.next to other node, don't forget to put pointer.next to a new variable in this way we won't lose it. In this problem, it's important to know whether head is None or not, we also need to know head.next for the reason that if this list only has 1 node.

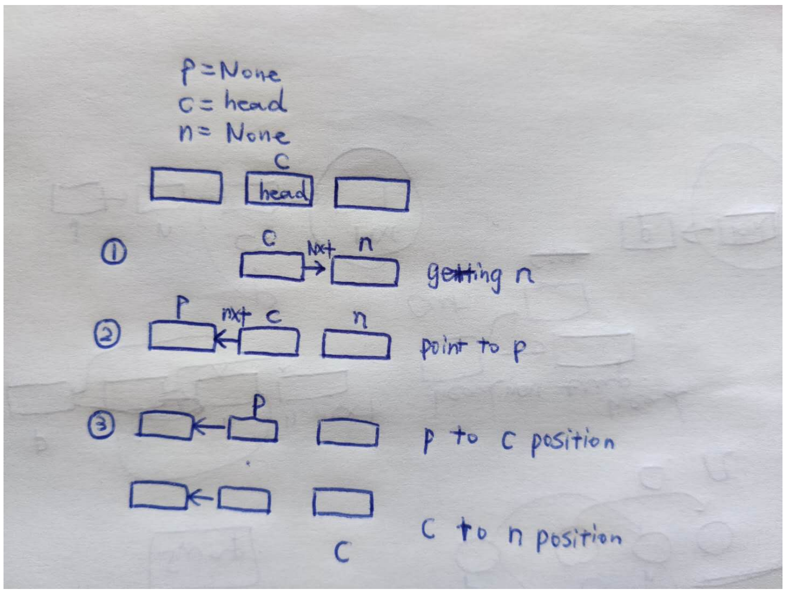


Figure 3.2: reverse list in iterative way

```
class Solution(object):
    def reverseList(self, head):
        if not head or not head.next:
            return head

        p = None
        c = head
        n = None

        while c:
            n = c.next
            c.next = p
            p = c
            c = n

        return p
```

For **recursive** way, reverseListBlog explained very clearly, the Figure shown below also illustrated the process of reverse a linked list in a recursive way.

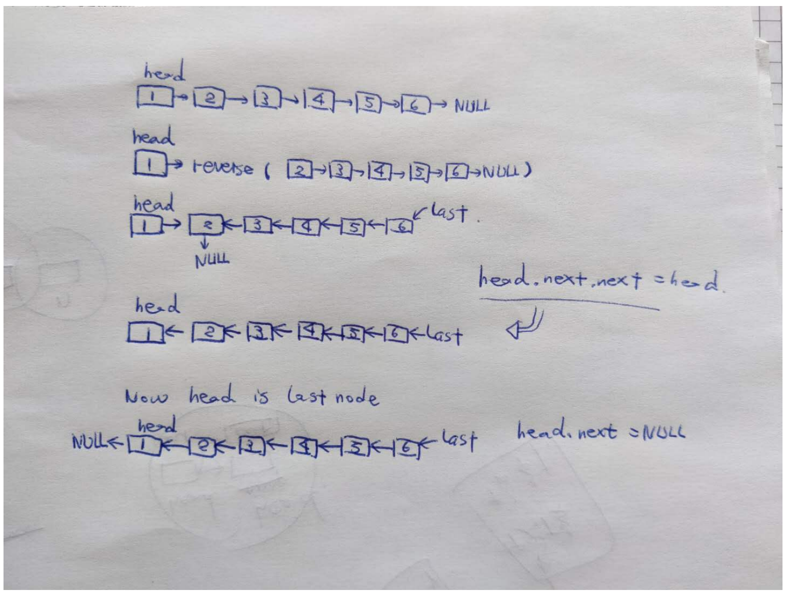

head
1 → 2 → 3 → 4 → 5 → 6 → NULL
head
1 → reverse (2 → 3 → 4 → 5 → 6 → NULL)
head
1 → 2 ← 3 ← 4 ← 5 ← 6 ← last .
NULL
head.next.next = head.
head
1 ← 2 ← 3 ← 4 ← 5 ← 6 ← last
Now head is last node
head
NULL ← 1 ← 2 ← 3 ← 4 ← 5 ← 6 ← last
head.next = NULL

Figure 3.3: reverse list in recursive way

```
def reverseList(self, head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if not head or not head.next:
        return head

    last = self.reverseList(head.next)
    head.next.next = head
    head.next = None
```

```
        return last
```

If we want to reverse the first N node, it is similar to the implementation of the recursive one, only need to know the successor, the successor is the one right after node N + 1, for example, 1-2-3-4-5-6-None, will return as 3-2-1 4-5-6, but in this case 1.next is not None anymore, it should be 4, so in recursive we need to know the node and assign that in a variable **successor**. The next is the Figure of Reverse N node, and code below the figure is the **Leetcode#92**

```
class Solution(object):
    successor = None
    def reverseBetween(self, head, m, n):
        def reverseN(head, n):
            global successor
            if n == 1:
                successor = head.next
                return head

            last = reverseN(head.next, n - 1)
            head.next.next = head
            head.next = successor

            return last

        if m == 1:
            # same as reverse first N node
            return reverseN(head, n)

        # forward as to the base case m = 1
        head.next = self.reverseBetween(head.next, m - 1, n - 1)
        return head
```

For **Leetcode#24**, swap nodes between nodes should could be solved with iterative way using 2 pointers n1 and n2, the graph and code is shown below.

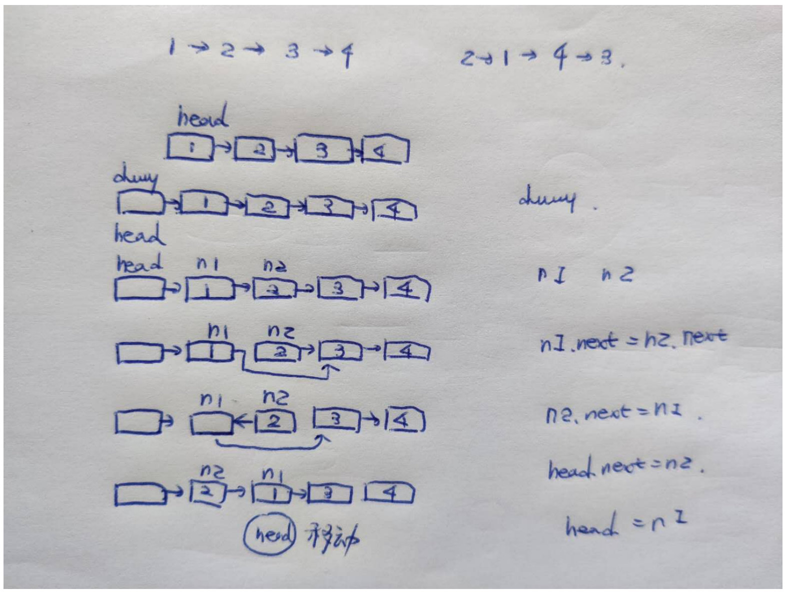


Figure 3.4: swap 2 nodes

```
def swapPairs(self, head):
    if not head or not head.next:
        return head
    dummy = ListNode(-1)
    dummy.next = head
    head = dummy

    while head.next and head.next.next:
        n1,n2 = head.next, head.next.next
        n1.next = n2.next
        n2.next = n1
        head.next = n2
        head = n1

    return dummy.next
```

3.2.3.3 Fast-slow pointer

Leetcode#141 Fast-slow pointer always used in check if a linked list has cycle, the fast pointer will go 2 step at a time while the slow pointer will go 1 step at a time. The initial step is to assign head to both of them, in while loop, the condition is while(fast.next), this is because if reach the second last node, fast.next is the last one, for the iteration after this will end.

```
def hasCycle(self, head):
    slow = head
    fast = head

    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next

        if fast == slow:
            return True

    return False
```

Leetcode#142, the only difference is to know the starting node of the cycle, the following graph shows after k step, slow and fast meet each other in the Meeting Point, the starting point should before m step of Meeting Point, after putting slow back to head, let slow and fast go 1 step at a time, after k - m step, they will meet at the Starting point. (Remeber there are 2 ways to go out of the while loop, the first one is break, the second one is normal running, then in that case there is no Cycle inside, we shoud return None instead).

```
def detectCycle(self, head):
    slow = head
    fast = head

    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next

        if slow == fast:
            break
    else:#No Cycle (out of loop not because of break)
        return None

    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return fast
```

Leetcode#876 is an easy one, same as above, when fast reaches the end, slow will be the middle node.

3.2.3.4 Advanced problems, not frequently asked questions

Leetcode#25 could be solved using reverseBetween, 2 passes in solving this, first is to count the length of the list, and the second is to iteratively call reverseBetween with corresponding index m and n. If the k is 2, then it is similar to **Leetcode#24**

```
def reverseKGroup(self, head, k):
    if (head is None or k == 1):
        return head

    # First, get the length of the list
    length = self.getCount(head)

    # Second pass, swap in groups.
    newH = head
    for i in range(1, length+1, k):
        if i+k-1 <= length:
            newH = self.reverseBetween(newH, i, i+k-1)

    return newH

def getCount(self,node):
    n = node
    count = 0
    while n:
        count+=1
        n = n.next
    return count
```

3.2.4 Stack

3.2.5 Queue

3.2.6 Deque

3.2.7 Tree

3.2.7.1 summary

- stack for doing the iterative form of traversal, the pattern of thinking should be reverse. Specifically, for inorder traversal, left-root-right, find left most first, but on the way to find left, stacking root also, since after getting left most, pop out order is left -¿ root. so stack.append(root) should be applied before finding left most node.**Memorization Trick:Y true not stack**

- For pre-order, root-left-right, unlike pop should be right before stack.append, since the stacking is reverse, stacking right first and then left branch. For postorder, the code is similar, left-right-root, we do it in a reverse way root-right-left, then the question becomes similar to pre-order, after getting res list, reverse it and then return.**Memorization Trick:Y stack**

- For Level-order(BFS), its better to use collections.deque(), which will save more time, the list.pop(0) have complexity of O(n).

- construct trees

- Getting the depth of a tree, normally using recursion, get left and right and consider returning a new value, which normally will be the max(left, right)+1 or other kinds of form. If there are other restriction in the tree depth, like finding the smallest, or finding compare the tree depth, then we to do something after getting and l and r.

- There are several problem set use both children but only return one child when doing the recursion, for example, #124, The definition of the path is **a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections**, means it could go through only left or right, but in the description Example, it shows that add both children only if the starting node of this path is their direct parent. If this is not the case, it can only return one of its children. So in the coding part. the trick is **res can append both children but only return one**. for #534 and #687 are also the same.

- PathSUM problem

- Go through the root or dose not go through the root? From root to the leaf, should give condition on the

Different kind of tree

- Binary Search Tree: the inorder traversal is sorted. Those 2 algorithms are bonded tightly

-

3.2.7.2 Traversal

Leetcode #94 is a classic **inorder** traversal of tree question, the following is the template of recursively do the inorder traversal.

```
def inorderTraversal(self, root):
    # recursive way
    res = []
```

```
        def inorder(root):
            # base case
            if not root:
                return root

            inorder(root.left)
            res.append(root.val)
            inorder(root.right)

        inorder(root)
        return res
```

The recursive way is shown below, the intuition behind the algorithm is that we want to get the leftmost child first in the res list, using stack here to push the previous nodes inside util we find the fist leftmost child (means it has no left child anymore). **Leetcode #173** is also using iterative way to inorderly traverse a Binary Search Tree.

```
    def inorderTraversal(self, root):
        # iterative way
        res = []   # the result oredr
        stack = [] #

        while True:
            while root:
                # find the left most
                stack.append(root)
                root = root.left
            # if there was no element inside stack, similar to if not root in recursi
            if not stack:
                return res

            # found the left most child
            left = stack.pop()
            res.append(left.val)
            root = left.right
```

Leetcode #589, Leetcode#114 is the **preorder** and **Leetcode #590** is the **postorder**.

```
    def preorder(self, root):
        # recursive way
        res = []

        def preorder1(root):
            if not root:
                return

            res.append(root.val)

            for c in root.children:
                preorder1(c)

        preorder1(root)
        return res
```

The following is the answer for #114, since the output should be a linked list tree, then the res.append should append input TreeNode instead of root.val, and assign the next element inside the list to the right node and the left node to None.

```
    def flatten(self, root):
        # preorder recursive way
        res =  []

        def preorder(root):
            if not root:
                return root

            res.append(root)
            preorder(root.left)
            preorder(root.right)

        preorder(root)

        for i in range(len(res) - 1):
            res[i].left = None
            res[i].right = res[i+1]

        return res
```

And the iterative way is as followed, after res.append, we need to reverse the way of the output, specifically it means that stack the one that needs to output later first, in this case it's the last child of the children, so we could simply use node.children[::-1] in this case, and use extend to get rid of the square braces problems.

```
    def preorder(self, root):
        # iterative way
        res = []

        if root == None:
            return res
        stack = []
        stack.append(root)

        while stack:
            node = stack.pop()
            if not node:
                continue
            # reverse the way to output, so stack in right first
            res.append(node.val)
            # stack.append(node.right)
            # stack.append(node.left)
            stack.extend(node.children[::-1])
        return res
```

The following is the graph shows how it works in general
The serialization of the preorder is also a problem set, for example in **leetcode #331**, the form of the preorder is always follows number # #.

```
    def isValidSerialization(self, preorder):

        #the leaves will always follow the form of number # #
        # only need to replace the last 3 element to # and iterate the list

        stack = collections.deque()

        pre = preorder.split(",")


        for i in pre:
            print(i)
            stack.append(i)

            while len(stack) >= 3 and stack[-1] == stack[-2] == '#' and stack[-3] !=
                stack.pop()
                stack.pop()
                stack.pop()
                stack.append('#')

        return len(stack) == 1 and stack.pop() == '#'
```

For postorder, the recursive way is the same as inorder and preorder except the order of append root.val, the following is the solution for **Leetcode #590**. The form is similar to the iterative form of preorder, but stacking node.left and then node.right, and reverse all the res list.

```
def postorder(self, root):
    # iterative way
    res = []
    if root == None:
        return res
    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()
        # if not node:
            # continue
        res.append(node.val)
        stack.extend(node.children[:])

    res.reverse()
    return res
```

recursive way for implementing the traversal is actually important in this section, since they are basic framework (or say template) to solve other kinds of problem.

```
def recursive(root):
    if not root:
        return
    # do what you want to do with root
    recursive(root.left)
    recursive(root.right)
```

In **Leetcode #100**, in order to know whether 2 tree are the same, in the recursion, first take out their root and compare. The recursion is also something will be coverred in DFS, where the fist line in the recursion are always those end case. In sameTree, the end condition is when 2 nodes are None, if not, then check if one of them is empty. if the structure are the same, the final condition is whether the values are the same. This ensemble the war defense, soldiers set up layer to defend enermies.

```
def isSameTree(self, p, q):
    # both of them are None, only one case with output True
    if p is None and q is None:
        return True
    #one of it is None
    if p is None or q is None:
        return False
    if p.val != q.val:
        return False

    return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

For **Leetcode #101**, same as isSameTree, only difference is change recursion to opposite node. self.isSameTree(p.left, q.right) and self.isSameTree(p.right, q.left)
For **Leetcode #104** finding the maximum depth of the tree, simply three lines could solve it. Same pattern as the traversal recursion. When root is None, return 0, otherwise recursively call the max (left and right) + 1.

```
def maxDepth(self, root):
    if not root:
        return 0
    return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

For minimum (**#111**) depth of the tree, after recursively calling the function, we need to know 2 situation, which is drawn below, when it comes to leave, 2 nodes like the graph should be counted as 2, so l + r + 1 = 2, while the other condition is similar to max.

```
def minDepth(self, root):
    if not root:
        return 0

    l = self.minDepth(root.left)
    r = self.minDepth(root.right)
    # be careful! if there ire only 2 nodes
    if l == 0 or r == 0:
        return l + r + 1
    else:
        return min(l, r) + 1
```

Same for **#110**, get hight using maxDepth function as in #104 and add one condition abs(l - r) ¡= 1 into the recursion call.
In the next subsection, we're going to discuss Level order traversal, this is also a sub pre-requisite for Search part BFS. The following code is the template for level order traversal(BFS).

```
def level_order_tree(root, result):
    if not root:
        return
    # using collections.deque()
    # avoid using list.pop(0) since the complexity is O(n)
    queue = collections.deque()
    queue.append(root)
    while queue:
        node = queue.popleft()
        # do somethings
        result.append(node.val)
```

```
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        return result
```

In most of the leetcode problem set, we need t output according to layers, so inside the while, we need to have a for loop and a newQueue which could do it in different layer . **#102** and **#107** are using the template below.

```
def level_order_tree(root):
    if not root:
        return
    q = [root]
    while q:
        # normally res will be here
        new_q = []
        for node in q:
            # do somethins with this layer nodes eg: 103, set flag to append in zigzag fash
            # classical condiction
            if node.left:
                new_q.append(node.left)
            if node.right:
                new_q.append(node.right)
        # replace as the new queue
        q = new_q
    # return things you want
    return xxx
```

#987 is a good problem set, it combines the level-order problem with coordinate questions. The first need to watch out is the form inside queue, should be (node, x, y), the order of the value and node is not matter, while inside res, we should put x and y in front since in the end of the while, res should be sorted in this way we'll could put those with the same coordinate in one cell from left to right. up to down, so y should be incremented instead of decremented as mentioned in the problem description.

```
    def verticalTraversal(self, root):
        res = []
        if not root:
            return res

        q = []
        #initial coordinates
        q.append((root, 0, 0))
        while q:
            node, x, y = q.pop()
            res.append((x, y, node.val))
            # no need to do it in differen layer
            # so no for loop here
            if node.left:
                q.append((node.left, x-1,y+1))
            if node.right:
                q.append((node.right, x+1,y+1))

        res.sort()
        print(res)
        # put root inside the res1 first
        res1 = [[res[0][2]]]
        # res2 = []
        for i in range(1, len(res)):
            if res[i][0] == res[i - 1][0]:
                res1[-1].append(res[i][2])
            else:
                res1.append([res[i][2]])

        return res1
```

#116 and **#117**

```
    def connect(self, root):
        """
        :type root: Node
        :rtype: Node
        """
        # Level-order way with iterative
        res = []
        if not root:
            return None

        q = collections.deque()
        q.append(root)

        while q:
            newQ = collections.deque()
            for i in range(len(q)):
                n = q.popleft()
                res.append(n.val)
                if n.left:
```

```
                newQ.append(n.left)
            if n.right:
                newQ.append(n.right)
        res.append('#')

        q = newQ

    a = Node(res[0])
    p = a
    for i in range(1, len(res)-1):
        if res != '#':
            b = Node(res[i])
        else:
            b = None
        a.next = b
        a = b
    return p
```

3.2.7.3 Binary Tree Pruning

Leetcode #814 is still a postorder, could use DFS(recursive way) to do it, since we need to know it from the leaves and then bottom up pruning the tree, the return value should be a node, the one to get the return value (condition) should be root.left and root.right. Trim it in the last step after all the coditions.

```
def pruneTree(self, root):
    if not root:
        return
    root.left = self.pruneTree(root.left)
    root.right = self.pruneTree(root.right)
    # only return node with 1 inside or not leaves
    return root if root.val == 1 or root.left or root.right else None
```

For **Leetcode #669**, trim it when compare to L and R and get left and right in the last condition.

```
def trimBST(self, root, L, R):

    if not root:
        return

    if root.val < L:
        return self.trimBST(root.right, L, R)

    elif root.val > R:
        return self.trimBST(root.left, L, R)

    else:
        root.left = self.trimBST(root.left, L, R)
        root.right = self.trimBST(root.right, L, R)
        return root
```

3.2.7.4 DFS related questions

A lot of problem sets could be solved in a recursive way. The followings are some problems that is typical recursion-based question. It is noticeable to know the nature of recursion, recursion is a structure of tree, every branch (or same iteration) will go through the deepest leave and if found no node after, then backtrack and return. So the first couple lines of DFS are always condition, the first line will also the last condition, for example in tree the first condition is always to know whether this node is empty or not to see if this recursion has reached the end of the tree, then next couple of conditions will be based on specific question set, the last is to recursively called the function

Leetcode #112 and **Leetcode #113** are among them, the difference of them is that in 112 there is only one answer while in 113 there are multiple answers, then we need a path + [root.left.val] inside the recursion, since we call the root.left.val which means we assume root.left is indeed exist, we can't wait until the next iteration to check whether root.left is existed, then in this round, it's also necessary to check if root.left is existed. The 2 templates below coud be used in any path sum related questions.

```
    def hasPathSum(self, root, s):

        if not root:
            return False

        if root.left == None and root.right == None and s == root.val:
            return True
        # either branch is okay, so OR is used here
        return self.hasPathSum(root.left, s - root.val) or self.hasPathSum(root.right

    def pathSum(self, root, s):
        res = []
        if not root:
            return res

        def helper(root, s, path):
            if not root:
                return
            if root.left == None and root.right == None and s == root.val:
                res.append(path)
                return

            if root.left:
                helper(root.left, s - root.val, path + [root.left.val])
            if root.right:
                helper(root.right, s - root.val, path + [root.right.val])

        helper(root, s, [root.val])

        return res
```

Leetcode#437 is also among them. But 437 has several answer and which may not from root to leave, which also need to recurse starting from every node below. then there are 2 recursion running below. First is the helper which did as the same as what we need above, and called pathSum twice and recursively call root.left and root.right inorder to start with a new child node to search.

```
    def pathSum(self, root, s):
        if not root:
            return 0
        return self.helper(root, s) + self.pathSum(root.left, s) + self.pathSum(root.right,

    def helper(self, root, s):
        if not root:
            return 0
        res = 0
        if s == root.val:
            res += 1

        res += self.helper(root.left, s - root.val)
        res += self.helper(root.right, s - root.val)
        return res
```

The next part is MAX PathSUM related questions: Use both children, but return one, these kind of question is a little bit tricky, but it can also be sloved using DFS way. For **Leetcode#124**, the graph has been shown below, we need to find the maximum path sum, the end condition is not go through the leaves, so we need no condition to stop util it reaches the end of the tree, but the tricky part is the result layer could be root.val+l+r while in recursion, we can only choose one of the path(biggest path) to search. And in case of negative number, l and r should be compared to 0.

```
def maxPathSum(self, root):
    self.res = float('-inf')

    def maxPath(root):
        if not root:
            return 0

        l = max(0, maxPath(root.left))
        r = max(0, maxPath(root.right))
        # final answer should add l + r + root.val
        self.res = max(l + r + root.val, self.res)
        # while this return will go the next layer, only need one path
        return max(l, r) + root.val

    maxPath(root)
    return self.res
```

leetcode #235 and **leetcode #236** is also part of these problems, like what we did in maxPath, divide(recursively called and get l and r first) and conquere (and then do something with the result l and r).

```
def lowestCommonAncestor(self, root, p, q):
    if any((not root,root == p, root ==q)):
        return root

    l = self.lowestCommonAncestor(root.left, p, q)
    r = self.lowestCommonAncestor(root.right, p, q)

    #means there are l and r with same root
    if l and r:
        return root
    # if not, return one of the branch which has and recursively call again
    else:
        return l if l else r
```

508

3.2.7.5 Construct Tree

The approach to construct tree is very simple, given the rule(BST, then mid is root)(max tree, build accoring to max in every recursive array), and slice the array for left branch and right branch inside the two recursive expression shown below.

```
def buildTree(self, preorder, inorder):
    if not preorder or  not inorder:
        return  None

    # firt elementin  preorder is root,  found the index in preorder and separate them
    root = TreeNode(preorder[0])
    index = inorder.index(preorder[0])
    # length of left branch in preorder is 1(root) to index + 1,
    root.left = self.buildTree(preorder[1:index+1], inorder[:index])
    root.right = self.buildTree(preorder[index+1:], inorder[index+1:])
    return root
```

3.2.7.6 LCA

These kinds of problem sets are a little bit difficult. 236 and 235 is the LCA problem, the recursion thinking pattern is to get l and r first, if found it, then could be return. since the return value is the definition of the function. But this process is after the recursion, the condition that could find the result is before the recursion, when the root is None, means not root, then return root, if the root is equal to p or q(remember this is after the recursion, root.left and root.right has been passed). if both returns a valid node which means p, q are in different subtrees, then root will be their LCA. if only one valid node returns, which means p, q are in the same subtree, return that valid node as their LCA.

```
    def lowestCommonAncestor(self, root, p, q):
        if not root or q == root or p == root:
            return root

        l = self.lowestCommonAncestor(root.left, p, q)
        r = self.lowestCommonAncestor(root.right, p, q)

        if l and r:
            return root
        else:
            return l if l else r
```

Iterative Way:

```
    def lowestCommonAncestor(self, root, p, q):
        pathP, pathQ = self.findpath(root, p), self.findpath(root, q)
        if pathP and pathQ:
            length = min(len(pathQ), len(pathP))
            ans = None
            for i in range(length):
                if pathP[i] != pathQ[i]:
                    break
                ans = pathP[i]
        return ans
```

```
def findpath(self, root, key):
    if not root:
        return
    stack = []
    lastVisit = None
    while stack or root:
        if root:
            stack.append(root)
            root = root.left
        else:
            tmp = stack[-1]
            if tmp.right and tmp.right != lastVisit:
                root = tmp.right
            else:
                if tmp.val == key.val:
                    return stack
                else:
                    lastVisit = stack.pop()
                    root = None
```

3.2.7.7 Tree serialization and de-serialization

#652 #297

3.2.7.8 Binary Search Tree

#98 To validate a binary search tree, normally BST-related questions could be solved using recursion, so this problem coud be solved in 2 ways. The first is give limit condition, condition is shown below.

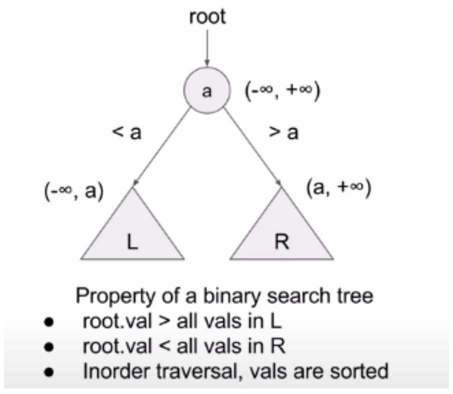


Figure 3.5: Validate Binary Search Tree

```
def isValidBST(self, root):
    return self.isvalid(root, float('inf'), float('-inf'))

def isvalid(self, root, max, min):
    if not root:
        return True

    if root.val >= max or root.val <= min:
        return False

    return self.isvalid(root.left, root.val, min) and \
           self.isvalid(root.right, max, root.val)
```

The second way is to use inorder, the inorder traversal of BST is sorted, so just to know in the recursion if prev is greater or equal to root, then return False, it is also noticeable that the assignment of prev should be after the first recursion. class Solution(object): def isValidBST(self, root): self.prev = None

def inorder(root): if not root: return True

if not inorder(root.left): return False

if self.prev and self.prev.val ¿= root.val: return False

self.prev = root

return inorder(root.right)

return inorder(root)

#701 combining the concept of binary search and build tree, very elegant solution.

```
def insertIntoBST(self, root, val):
    if not root:
        return TreeNode(val)
    if val > root.val:
        root.right = self.insertIntoBST(root.right, val)
    if val < root.val:
        root.left = self.insertIntoBST(root.left, val)
    return root
```

# 501 is a very good problem, to find the highest frequency element inside the Binary search tree, the first solution is to use collections.Counter() to do that, and there are possibilities that not only 1 highest frequency element, that is why the return should be a list.

```
def findMode(self, root):
    if not root: return []
    self.count = collections.Counter()
    self.inorder(root)
    freq = max(self.count.values())
```

```
        res = []
        for item, c in self.count.items():
            if c == freq:
                res.append(item)
        return res

    def inorder(self, root):
        if not root: return
        self.inorder(root.left)
        self.count[root.val] += 1
        self.inorder(root.right)
```

#450,

- The deleted node has no left tree, return right tree
- The deleted node has no right tree, return left tree
- Has both left and right tree, to way to do it
  - Search the min in right tree, swap the nodes and recursively delete the key again
  - Search the max in the left tree, swap the nodes and recursively delete the key again

```
    def deleteNode(self, root, key):
        if not root: return root
        if root.val == key:
            # if root is a leave
            if not root.right:
                return root.left
            else: # both left and right exist
                  # set the min of right tree to be the new node
                right = root.right
                while right.left:
                    right = right.left
                root.val, right.val = right.val, root.val

        root.left = self.deleteNode(root.left, key)

        root.right = self.deleteNode(root.right, key)

        return root
```

3.3 Advanced Data Structure

3.3.1 Priority Queue

3.3.2 Trie

3.3.3 Segment Tree

3.4 Search

3.4.1 Backtrack

Summarize from labladong:

- subset: start
- combination: start
- permutation used
- bfs:
- dfs:

Backtracking is the traversal of decision tree in essence. There are 3 things:

- Path: decisions have been made
- Choice list: the choices you can choose now
- End condition: reach to the end of the decision tree

General framework of the backtracking is to do **recursion in for loop, do a choice before the recursion and undo the choice after the recursion**

```
res = []
def backtrack(path,choice_list):
    if end_condition:
        result.append(path)
        return

    for c in choice_list:
        do the choice (select the choice)
        backtrack(path, choice_list)
        undo the choice c (remove the choice from choice_list)
```

3.4.1.1 combination

Input 2 number n and k, output the all combinations of k number in [1..n]. This is also a backtracking problem, but k limit the tree depth, and n limit the with of the tree. Which means the algorithm is similar to subset problem, the only difference **is that k limit the depth, so the end condition should take k into account and end the recursion**. Other than this, combination and subset problem should include **start** index to indicate that the element used before can not be used. There are 2 cases.

- backtrack(path + [something], i): means if combining [1,2,3,4], i = 1 then element 2 can be used in the next round but 1 cannot be used (diffrent from permutation)
- backtrack(path + [something], i + 1): means if combining [1,2,3,4], i = 1 then element 2 can not be used in the next round but 1 cannot be used, it could only start from element 3(index = 2)

Figure 3.6 shown below shows the tree like search process, in some cases, the combination will have constraints like there is **duplicate** inside the list choices but the output cannot have duplicate result, this Figure below(lowest tree) shows it will have same result when it contains duplicate, then an expression **if i > start and c[i] == c[i - 1] continue** should be used, and the choice list should be **sorted** beforehand. Also path + [i] is a technique here work as DFS, the traditional way is to path.append() and path.pop(). #17 #39 #40 are the combination problem, the template below is the solution of #40 Combination Sum II.

```python
def combinationSum2(self, c, t):
    res = []
    if c is None or t == 0:
        return res

    def backtrack(path, target, start):
        if target == 0:
            res.append(path)
            return

        for i in range(start, len(c)):
            if c[i] > target:
                continue
            if i > start and c[i] == c[i - 1]:
                continue
            backtrack(path + [c[i]], target - c[i], i + 1)

    c.sort()
    backtrack([], t, 0)
    return res
```

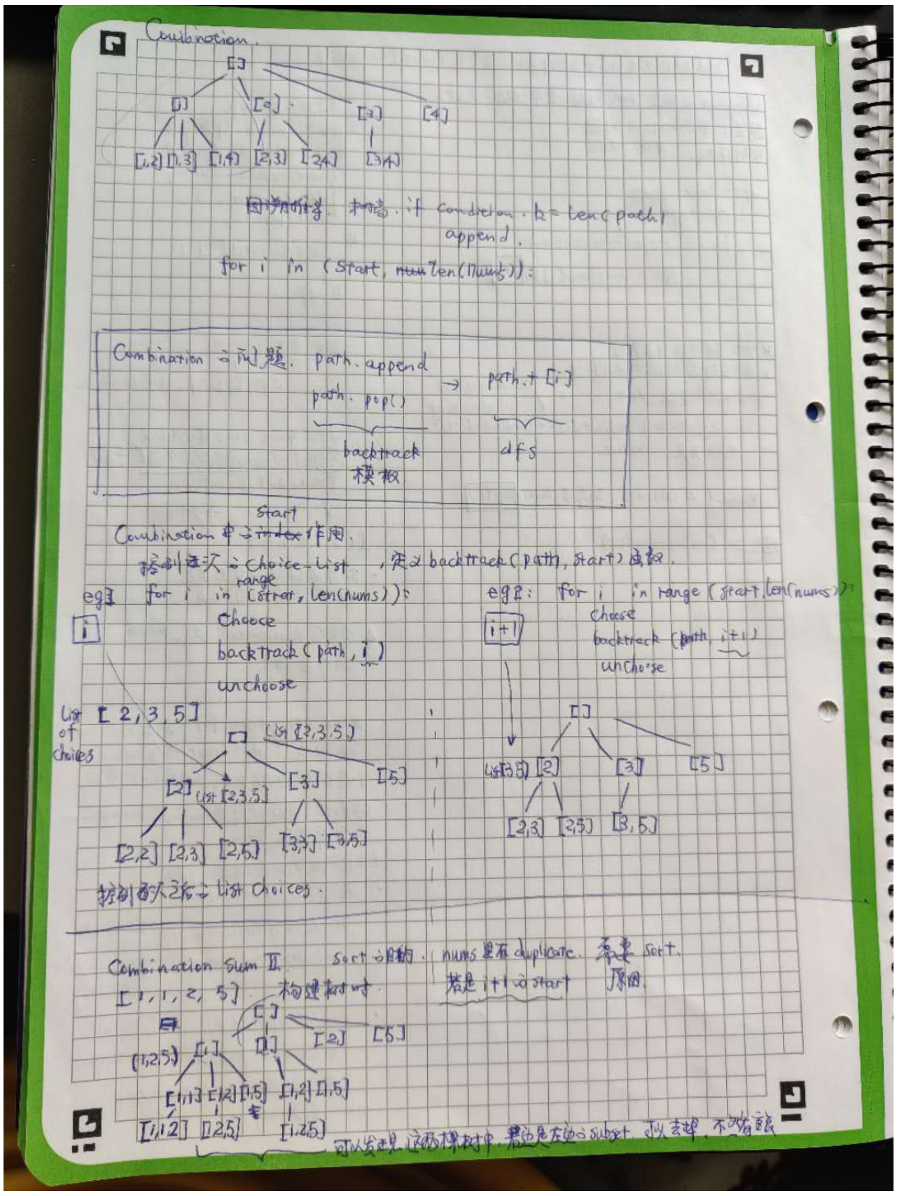


Figure 3.6: Combination

3.4.1.2 subset

concept of these questions: given an array, return its all subsets. eg: nums = [1,2,3], the subsets are [[],[1],[2],[3],[1,3],[2,3],[1,2],[1,2,3]], the order dosen't matters. The question is similar to combination, the only difference is that it doesn't need to have any specific rule to append the path into res.

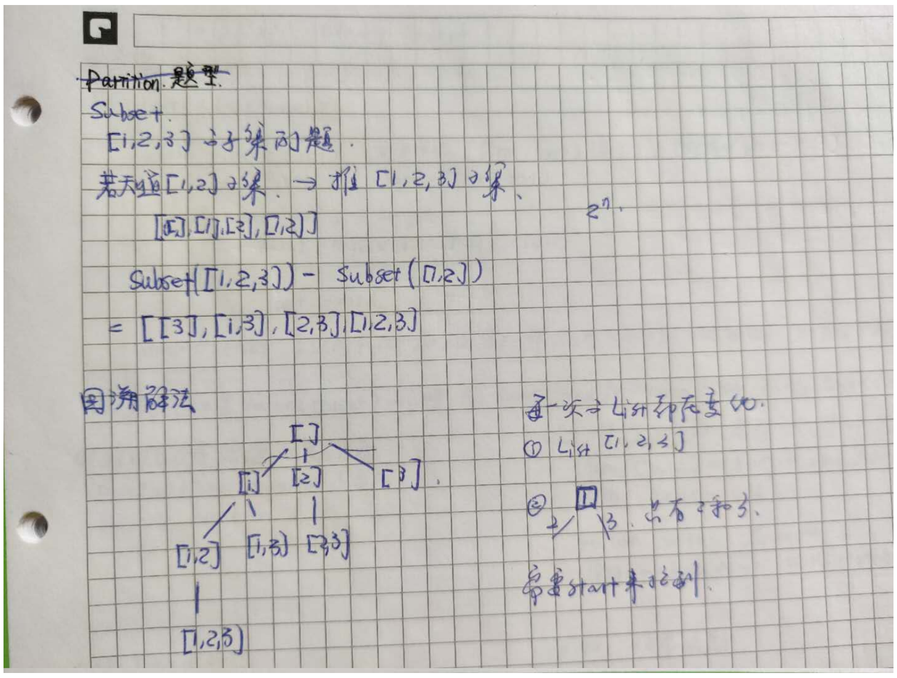


Figure 3.7: subset

3.4.1.3 permutation

The only difference with subset and combination is that the number before could be used, so be careful when doing this, we only need a 'used' array to denote whether the element have been used in this recursion. The followng is the solution for #47

```
def permuteUnique(self, nums):
    res = []
    if nums == None:
        return res
    used = [0] * len(nums)
    def backtrack(path):
        if len(path) == len(nums):
            res.append(path)
            return

        for i in range(len(nums)):
            if i > 0 and nums[i] == nums[i - 1] and used[i - 1]:
                continue
            if used[i]:
                continue
            used[i] = 1
            backtrack(path + [nums[i]])
            used[i] = 0

    nums.sort()
    backtrack([])
    return res
```

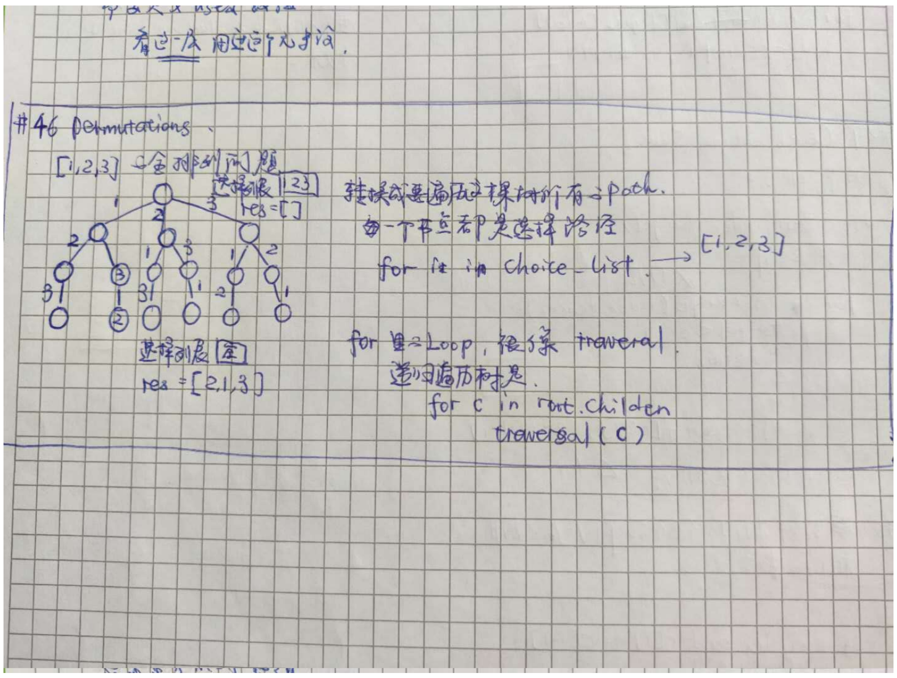


Figure 3.8: Permutation

3.4.2 BFS

The template of BFS is similar to level order traversal of a binary tree, using queue to appedn root and while loop the root, since we need to get the node level by level, so the size of nodes inside queue should be gained in order to for loop all the nodes inside queue. BFS is a typical **Start** to **Target** problem, and always is to find the **Shortest Path from Start to the Tartget**. Recall in level order traversal there is no end condition in for loop, while in BFS, if we found the **Target**, then we should return the step. A typical return value is the value(num of levels to get to the target). Other than that, since this is not a tree anymore, the search direction is not stricted in downward(left node or right node), it could search all of the adjacent nodes, so a visited **set** should be initialized to ensure the node has not been visted yet. BFS is usually used for solving the least step to solve some search based problem.

126 word ladder is a typical BFS problem. But in typical BFS problem, normally inside the for loop, we will see if the node is in visited, if it is in visied, then stop this loop with **continue**, while in this case we want to find the node which is in the visited,

so those are not in the visited should be ignored. So strictly speaking, this **set** should be declared as wordListSet instead of visited.

```
def ladderLength(self, beginWord, endWord, wordList):
    visted = set(wordList)
    q = collections.deque()
    q.append(beginWord)
    if endWord not in visted: return 0
    step = 0

    while q:
        qSize = len(q)
        step += 1
        for k in range(qSize):
            node = q.popleft()
            for i in range(len(node)):
                for j in "abcdefghijklmnopqrstuvwxyz":
                    word = node[:i] + j + node[i+1:]
                    if endWord == word and word in visted:
                        return step+1

                    if word not in visted:
                        continue

                    if word in visted and word != node:
                        visted.remove(word)
                        q.append(word)
    return 0
```

#752 Open the lock, the first code ”0000” should also be put into the deadends set to avoid go back to the origin lock set. simple, but the point of the circle number is important to be rememberred. 0 to 9 and 9 to 0, which could be done by mod 10.

```
def openLock(self, deadends, target):
    if "" == target or "0000" in deadends or target in deadends:
        return -1

    q = collections.deque()
    q.append("0000")

    visited = set(deadends)
    visited.add("0000")

    step = 0
```

```
        while q:
            step += 1
            qSize = len(q)

            for i in range(qSize):
                node = q.popleft()
                for j in range(len(node)):
                    for k in (1, -1):
                        num = (int(node[j]) + k) % 10
                        new = node[:j] + str(num) + node[j+1:]

                        if new == target:
                            return step

                        if new in visited:
                            continue

                        q.append(new)
                        visited.add(new)
        return -1
```

#542 01 Matrix, get every step into the dist function. the interesting thing here is when search in 4 different directions, we could use dirs = [0,-1,0,1,0] to reduce the amount of code we need to write, simply loop in range(4) and add the first to x and add the second to y. This solution below is the one that will not exceed the time limit

```
    def updateMatrix(self, matrix):
        n = len(matrix)
        m = len(matrix[0])

        q = collections.deque()
        dist = [[float("inf") for i in range(m)] for j in range(n)]

        for i in range(n):
            for j in range(m):
                if matrix[i][j] == 0:
                    dist[i][j] = 0
                    q.append((i, j))

        dirs = [0,-1,0,1,0]
        while q:
            x, y = q.popleft()
            for i in range(4):
                a = x + dirs[i]
                b = y + dirs[i+1]
                if  0<=a<n and 0<=b<m and dist[a][b] - dist[x][y] > 1:
                    dist[a][b] = dist[x][y] + 1
                    q.append((a, b))

        return dist
```

#675 Cut Off Trees for Golf Event.

```
 def cutOffTree(self, f):
        if f == None:
            return -1
        h = []
        # sort the height
        m = len(f)
        n = len(f[0])
        for i in range(m):
            for j in range(n):
                if f[i][j] != 0:
                    h.append((f[i][j], i, j))

        def bfs(f, sx, sy, tx, ty):
            q = collections.deque()
            q.append((sx, sy))
            visited = [[0 for i in range(n)] for j in range(m)]

            step = 0
            dirs = [0, -1, 0, 1, 0]
            while q:
                qSize = len(q)

                for i in range(qSize):
                    (x, y) = q.popleft()
                    if x == tx and y == ty:
                        return step
                    for j in range(4):
                        nx = x + dirs[j]
                        ny = y + dirs[j + 1]

                        # Out of bound or unwalkable
                        if (nx < 0 or nx == m or ny < 0 or ny == n or not f[nx][ny] o
                            continue
                        q.append((nx, ny))
                        visited[nx][ny] = 1
                step += 1

            return float('inf')

        # sort the height
        h.sort()
        sx = 0
        sy = 0
        ts = 0

        # loop the height an call BFS
        for i in range(len(h)):
            tx = h[i][1]
            ty = h[i][2]

            s = bfs(f, sx, sy, tx, ty)
            if s == float('inf'):
                return -1

            ts += s
            sx = tx
            sy = ty

        return ts
```

3.4.3 DFS

DFS is backtracking problem in essence, for #301, this is a hard question, the template is the same as the backtrack problem, the difference is that we need to know how many l and r should be deleted(kinda like target in this case), for every element is list_choice, just simply remove the element at this level and continue the recursion. curr = s[:i] + s[i+1:] will do this action. While the way to verified if a sequence of parentheses are valid is shown in the section 3.1.2.

```
    def removeInvalidParentheses(self, s):
        res = []
        if s == "":
            return [""]
        l = 0
        r = 0
        for i in s:
            if i == '(':
```

```
            l += 1
        if i == ')' and l == 0:
            r += 1
        elif i == ')':
            l -= 1

#2. if a string is valid
def valid(s):
    l = 0
    r = 0
    for i in s:
        if i == '(':
            l += 1
        if i == ')' and l == 0:
            r += 1
        elif i == ')':
            l -= 1
    return l == 0 and r == 0

def backtrack(s, l, r, start):
    if l == 0 and r == 0 and valid(s):
        res.append(s)
        return

    for i in range(start, len(s)):
        if i > start and s[i] == s[i - 1]:
            continue
        # if s[i] is parenthesis
        if s[i] == '(' or s[i] == ')':
            curr = s[:i] + s[i + 1:]
        # start from right parenthesis
            if r > 0:
                # start is i because i is removed, the i+1 now becomes
                # i, so start again with i instead of i + 1
                backtrack(curr, l, r-1, i)
            elif l > 0:
                backtrack(curr, l-1,r, i)
backtrack(s, l, r, 0)
return res
```

For #22, same backtracking problem, but the key thing is to understand for loop in backtracking, in this case, there is no need to use for loop, for loop only used in the situation that there are several list choices, in this case, the choices are left or right parentheses, but the actions are taken in different condition, which means it has all choices made already.

```
def generateParenthesis(self, n):
    res = []
    if n == 0:
        return [""]

    def backtrack(path, l, r):
        if l > r:
            return

        if l == 0 and r == 0:
            res.append(path)
            return
        if l > 0:
            backtrack(path + '(', l-1,r)
        if r > 0:
            backtrack(path + ')', l,r-1)
    backtrack("", n, n)
    return res
```

3.4.3.1 Sudoku, N-Queue

#37 Sudoku, this is similar to permutation, this is a brute force DFS, the used array have 3, cols, rows and boxs, the only thing need to be remember is the when found 1 solution, it could return True already, so it won't search other results. And the index of box is also important in this case x*3 + y.

```
def solveSudoku(self, board):
    if not board:
        return
    # initialize the board

    rows = [[0 for i in range(10)] for j in range(9)]
    cols = [[0 for i in range(10)] for i in range(9)]
    box = [[0 for i in range(10)] for j in range(9)]

    for i in range(9):
        for j in range(9):
            if board[i][j] != '.':
                a = int(board[i][j])
                rows[i][a] = 1
                cols[j][a] = 1
                box_x = i // 3
                box_y = j // 3
                box[box_x*3 + box_y][a] = 1

    def backtrack(x, y):
        if x == 9:
            return True

        ny = (y + 1) % 9
        if ny == 0:
            nx = x + 1
        else:
            nx = x

        if board[x][y] != '.':
            return backtrack(nx, ny)


        # backtracking
        for i in range(1, 10):
            box_x = x // 3
            box_y = y // 3
            box_index = box_x * 3 + box_y
            if not rows[x][i] and not cols[y][i] and not box[box_index][i]:
                # take actions
                rows[x][i] = 1
                cols[y][i] = 1
                box[box_index][i] = 1
                board[x][y] = str(i)

                if backtrack(nx, ny):
                    return True

                board[x][y] = '.'
                rows[x][i] = 0
                cols[y][i] = 0
                box[box_index][i] = 0
        return False

    backtrack(0, 0)
```

#51 N-Queue, the ‘used’ array also have 3, rows, diag1 and diag2, in this case, the bracktrack parameter x means cols(rows is also fine). For Sudoku, the choice list have 1 to 9, so for in range(1, 10), but in N-Queue, the choice now becomes cols, so only for in range(n). and we can get corresponding index of those ‘used’ array but only one index 'i', so rows, diags and diag2 is only a one dimentional array.

```
def solveNQueens(self, n):
    res = []
    if n == 0:
        return res

    rows = [0] * n
    diag1 = [0] * (2*n - 1)
    diag2 = [0] * (2*n - 1)

    #initialize the board
    board = [['.' for _ in range(n)] for _ in range(n)]

    def backtrack(x):
        if x == n:
            res.append(["".join(i) for i in board])
            return

        for i in range(n):
            if not rows[i] and not diag1[x+i] and not diag2[x - i + n - 1]:
                rows[i] = 1
                diag1[x+i] = 1
                diag2[x - i + n - 1] = 1
                board[x][i] = 'Q'

                backtrack(x+1)

                rows[i] = 0
                diag1[x+i] = 0
                diag2[x - i + n - 1] = 0
                board[x][i] = '.'
    backtrack(0)
    return res
```

# Word Search, same backtracking, but be careful when used this element in the board and set it to nil and set it back after the backtrack. Otherwise this element might be used again in the next round.

```
def exist(self, board, word):
    if word == "":
        return True
    m = len(board)
    n = len(board[0])
    def backtrack(x, y, word):
        if len(word) == 0:
            return True

        if x < 0 or x >= m or y < 0 or y >= n or board[x][y] != word[0]:
            return False

        t = board[x][y]
        board[x][y] = '#'
        res= backtrack(x+1, y, word[1:]) or backtrack(x, y+1, word[1:]) or backtra
        board[x][y] = t
        return res
    for i in range(m):
        for j in range(n):
            if backtrack(i, j, word):
                return True

    return False
```

3.4.3.2 DFS+BFS

#934 Shortest Bridge, this is a very good problem because it covers both DFS and BFS, to find the shortest path of 2 island, the first step is to find one of the island and marked it as diffrent element(2 in my case), because from the perspective of finding the same 1 in the deepest path, this is something DFS usually dose, so dfs is used here to search an island thoroughly, and the same time enqueue all of the node found in dfs, this is kinda like enqueue all node in the same level(From the perspective of another island).

The second step is to use classical BFS, the node marked as 2 could be re-used as 'visited' set in BFS.

```
def shortestBridge(self, A):
    # dfs and mark an island to 2
    m = len(A)
    n = len(A[0])

    q = collections.deque()
    # dfs
    def backtrack(x, y):
        # end condition
        if x == m or x < 0 or y == n or y < 0 or A[x][y] != 1:
            return
        A[x][y] = 2
        q.append((x, y))
```

```
            for a, b in ((x+1, y), (x-1, y),(x, y+1),(x, y-1)):
                backtrack(a, b)

        Found = False
        for i in range(m):
            for j in range(n):
                if A[i][j] == 1:
                    backtrack(i, j)
                    Found = True
                    break
            if Found == True:
                break


        # bfs
        dirs = [0, -1, 0, 1, 0]
        step = 0
        while q:
            qSize = len(q)
            for i in range(qSize):
                x, y = q.popleft()
                for j in range(4):
                    a = x + dirs[j]
                    b = y + dirs[j + 1]

                    if a == m or a < 0 or b == n or b < 0 or A[a][b] == 2:
                        continue

                    if A[a][b] == 1:
                        return step

                    A[a][b] = 2
                    q.append((a, b))

            step += 1

        return -1
```

3.5 Binary Search

The Binary Search Algorithm is very easy, but the detail is very important, lower bound, upper bound, return mid + 1 or mid - 1. The open interval and closed interval are also a question that needs to be remember.

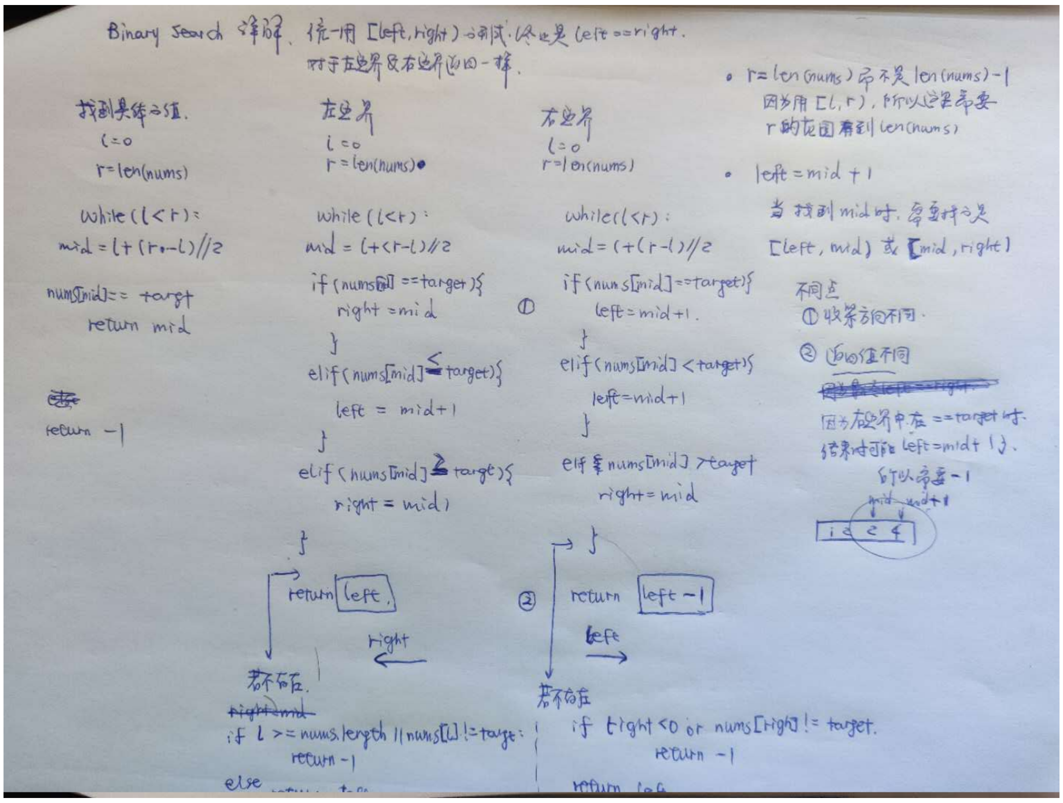
The template for the Binary Search is as follows:

```python
# return the first value that is not less than value in [first, last)
def lower_bound(array, first, last, value):
    while first < last:
        mid = first + (last - first) // 2
        if array[mid] < value: first = mid + 1
        else: last = mid
    return first # last is also fine
```

There are 2 cases here:

First is to find lower bound, which means to find x >= value or x > value, the index of x is the minimum. Use [first, last) interval.

Second is to find the upper bound, which is to find the x ¡ value or x <= value, where the index of x is the largest.



Binary Search 详解. 统一用 [left, right) 的形式. 终止是 left == right.
对于左边界及右边界返回一样.

找到具体的值.

```
l=0
r=len(nums)

while(l<r):
mid = l+(r-l)//2

nums[mid]== target
    return mid

return -1
```

左边界

```
l=0
r=len(nums)

while(l<r):
mid = l+(r-l)//2
if(nums[mid] ==target){
    right =mid
}
elif(nums[mid]<target){
    left = mid+1
}
elif(nums[mid]>target){
    right = mid)
}
return left
```

right ←

若不存在.
~~right=mid~~

```
if l >= nums.length || nums[l] != target:
        return -1
else
```

①

②

右边界

```
l=0
r=len(nums)

while(l<r):
mid = l+(r-l)//2
if(nums[mid]==target){
    left=mid+1.
}
elif(nums[mid]<target){
    left=mid+1
}
elif nums[mid] >target
    right=mid
}
return left-1
```

left →

若不存在

```
if right<0 or nums[right] != target.
        return -1
return
```

- r=len(nums) 而不是 len(nums)-1
  因为用 [l, r), 所以这里需要
  r 的范围看到 len(nums)
- left = mid + 1
  当找到 mid 时, 需要转到是
  [left, mid) 或 [mid, right)

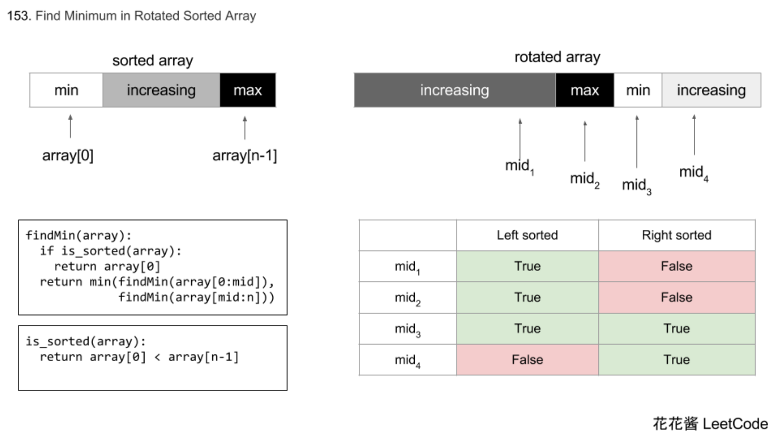
不同点
① 收缩方向不同.
② 返回值不同
~~[illegible]~~
因为右边界中在 ==target 时.
结束时可能 left=mid+1}.
所以需要 -1

Figure 3.9: peak number

leetcode #33 want to find a value inside a rotated array, this actually could be solved by using indexof but since the problem should be solved in nlogn, then binary search should be used here. The important part is **to know which part is strictly increasing seuqnce**, for example, a sequence is 4,5,6,7,0,1,2. say if we want to find value 0, if mid is in 7, then we simply compare mid with r and l to see which part is sorted, obviously, 4,5,6,7 is sorted, while 7,0,1,2 is not sorted, after know which part is sorted, just simply compare it with **target to see whether target is in this fields**. if not, then shrink the mid. Remember we need to compare our mid target with the right value, so we will use closure [l,r] instead of [l,r) in this one.

```
def search(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: bool
    """
    N = len(nums)
    l, r = 0, N - 1
    while l <= r:
        while l < r and nums[l] == nums[r]:
            l += 1
        mid = l + (r - l) / 2
        if nums[mid] == target:
            return mid
        if nums[mid] >= nums[l]:
            if nums[l] <= target < nums[mid]:
                r = mid - 1
            else:
                l = mid + 1
        elif nums[mid] <= nums[r]:
            if nums[mid] < target <= nums[r]:
                l = mid + 1
            else:
                r = mid - 1
    return -1
```

in leetcode #153, try to find the minimum of a rotated array, just use recursion+binary search to do that.

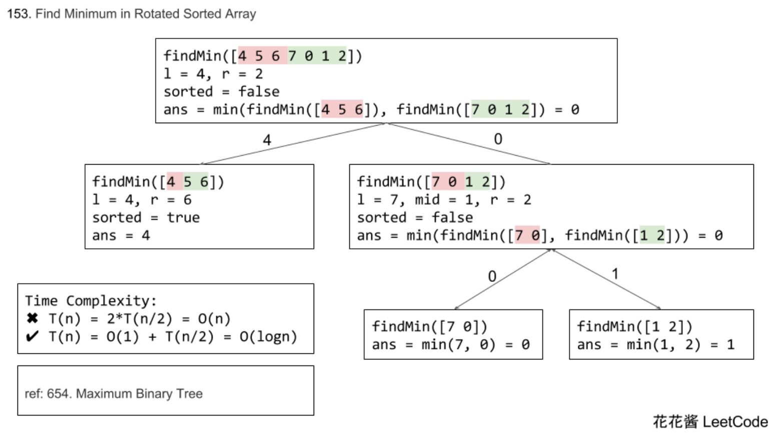


```
findMin(array):
  if is_sorted(array):
    return array[0]
  return min(findMin(array[0:mid]),
             findMin(array[mid:n]))
```

```
is_sorted(array):
  return array[0] < array[n-1]
```

	Left sorted	Right sorted
mid_1	True	False
mid_2	True	False
mid_3	True	True
mid_4	False	True

Figure 3.10: Minimum number of Rotated array 1



```
findMin([4 5 6 7 0 1 2])
l = 4, r = 2
sorted = false
ans = min(findMin([4 5 6]), findMin([7 0 1 2]) = 0
```

```
findMin([4 5 6])
l = 4, r = 6
sorted = true
ans = 4
```

```
findMin([7 0 1 2])
l = 7, mid = 1, r = 2
sorted = false
ans = min(findMin([7 0], findMin([1 2])) = 0
```

```
findMin([7 0])
ans = min(7, 0) = 0
```

```
findMin([1 2])
ans = min(1, 2) = 1
```

```
Time Complexity:
✖ T(n) = 2*T(n/2) = O(n)
✔ T(n) = O(1) + T(n/2) = O(logn)
```

ref: 654. Maximum Binary Tree

Figure 3.11: Minimum number of Rotated array 1

```
def findMin(self, nums):
    def bsMin(nums, l, r):
        # bs, end condition is l==r(only 1 element), if 2 left, l+1==r
        if l + 1 >= r:
            return min(nums[l], nums[r])

        # sorted
        if nums[l] < nums[r]:
            return nums[l]

        mid = l + (r - l) // 2
        return min(bsMin(nums, mid + 1, r), bsMin(nums, l, mid))
    l = 0
    r = len(nums) -1
    return bsMin(nums, l, r)
```

leetcode #162 peak number, need to find a peak number in an array, use 2 pointers in this case. The process is shown below.

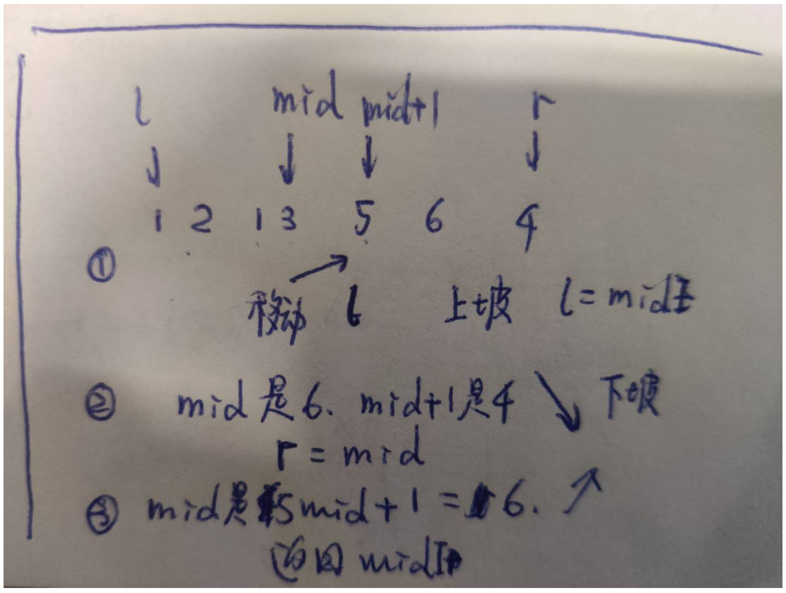


Figure 3.12: peak number

```
    def findPeakElement(self, nums):
        l = 0
        r = len(nums) - 1
        while l < r:
            mid = l + (r - l) // 2
            mid1 = mid + 1

            if nums[mid] < nums[mid1]:
                l = mid1
            else:
                r = mid

        return l
```

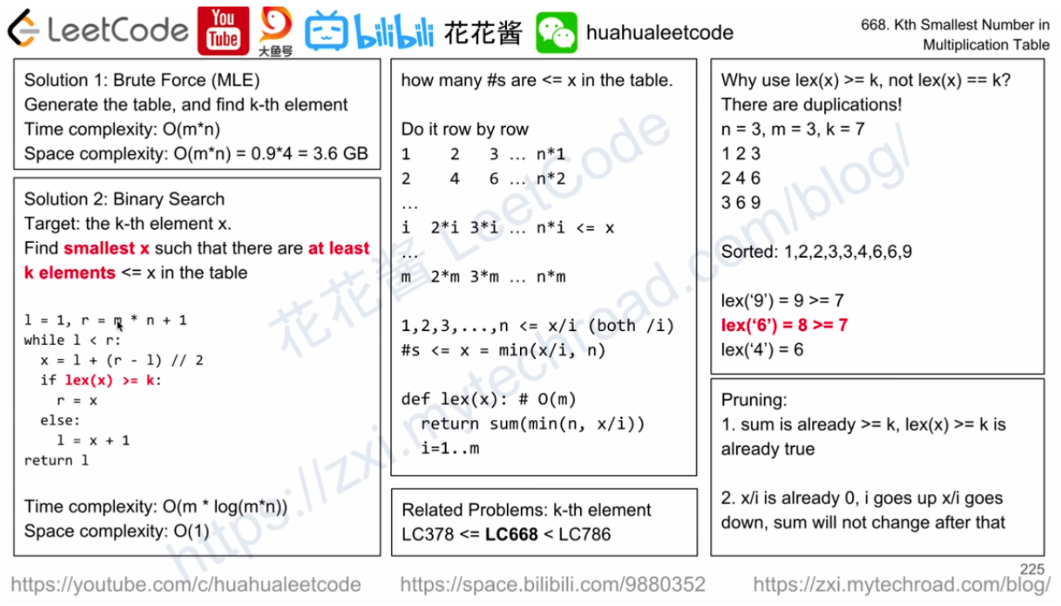
#69 upper bound, be careful that the l should be 1 instead of 0, the reason to find upper bound instead of lower bound is that **to find the first that is bigger than the result or say the rightest one that close to the result**

```
def mySqrt(self, x):
    l = 1
    r = x

    while l < r:
        mid = l + (r - l) // 2
        if mid ** 2 == x:
            return mid
        elif mid ** 2 > x:
            r = mid
        else:
            l = mid + 1

    return l - 1
```

#378 Find the smallest x(lower bound Binary Search), such that there are k elements that are smaller or equal to x(upper bound).

```
def kthSmallest(self, matrix, k):
    if not matrix:
        return -1

    l = matrix[0][0]
    r = matrix[-1][-1] + 1

    while l < r:
        mid = l + (r - l) // 2

        loc = sum(bisect_right(a, mid) for a in matrix)

        if loc == k:
            r = mid
        elif loc < k:
            l = mid + 1
        else:
            r = mid

    return l
```

#668, it is a little bit different from the #378, the matrix is just the multiplication of col and row, if use the way above(bisect_right) will cause memory limit, so instead of using sum over the bisect_right, we use a function called lex, for every rows(m), compare the col with mid/i, i is every row here, we take advantage of the fact that each cell in matrix is actually just a multiplication, mid/i can calculate the boundary in this line, for example, [1,2,3],[2,4,6],[3,6,9], the first mid will be 5, then when scan the first row, n=3, $mid/1 = 5$, so in the first line there are min(3,5) 3 numbers that is smaller than the mid, for the second line, m=2, mid=5, n=3, $mid//m = 2.5$, then in this case, there are 2 numbers that is smaller than 5 (2 and 4). The following slide shows the detail reasoning.



LeetCode 花花酱 huahualeetcode

668. Kth Smallest Number in Multiplication Table

Solution 1: Brute Force (MLE)
Generate the table, and find k-th element
Time complexity: O(m*n)
Space complexity: O(m*n) = 0.9*4 = 3.6 GB

Solution 2: Binary Search
Target: the k-th element x.
Find **smallest x** such that there are **at least k elements** <= x in the table

```
l = 1, r = m * n + 1
while l < r:
  x = l + (r - l) // 2
  if lex(x) >= k:
    r = x
  else:
    l = x + 1
return l
```

Time complexity: O(m * log(m*n))
Space complexity: O(1)

how many #s are <= x in the table.

Do it row by row

```
1    2    3 … n*1
2    4    6 … n*2
…
i  2*i 3*i … n*i <= x
…
m  2*m 3*m … n*m

1,2,3,…,n <= x/i (both /i)
#s <= x = min(x/i, n)

def lex(x): # O(m)
  return sum(min(n, x/i))
  i=1..m
```

Related Problems: k-th element
LC378 <= **LC668** < LC786

Why use lex(x) >= k, not lex(x) == k?
There are duplications!
n = 3, m = 3, k = 7
1 2 3
2 4 6
3 6 9

Sorted: 1,2,2,3,3,4,6,6,9

lex('9') = 9 >= 7
lex('6') = 8 >= 7
lex('4') = 6

Pruning:
1. sum is already >= k, lex(x) >= k is already true

2. x/i is already 0, i goes up x/i goes down, sum will not change after that

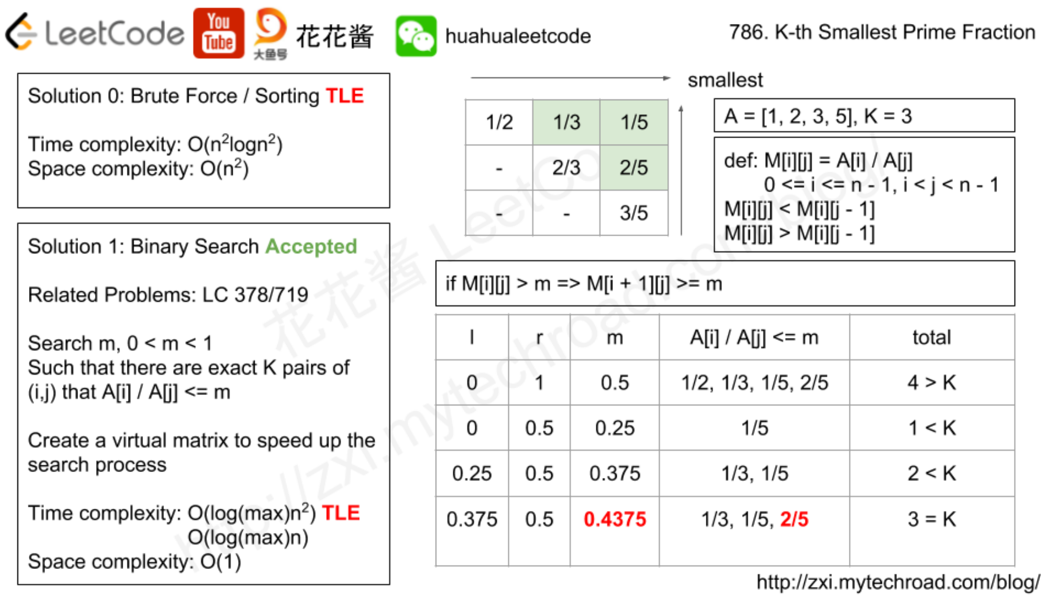
225

https://youtube.com/c/huahualeetcode https://space.bilibili.com/9880352 https://zxi.mytechroad.com/blog/

Figure 3.13: Kth Smallest Number in Multiplication Table (From huahua)

```
def findKthNumber(self, m, n, k):
    l = 1
    r = m*n+1

    def lex(mid, m, n):
        count = 0
        for i in range(1, m+1):
            count += min(n, mid // i)
        return count

    while l < r:
        mid = l + (r - l) // 2
        loc = lex(mid, m, n)

        if loc >= k:
            r = mid
        else:
            l = mid + 1
        return l
```

#786, a hard question, should take advantage of the property of the matrix, for each row, found the first col that is $A[i]/A[j] <= m$, in other words, $A[i] < A[j] * m$, so the rest will all smaller. The detail is shown below.



LeetCode 花花酱 huahualeetcode

786. K-th Smallest Prime Fraction

Solution 0: Brute Force / Sorting TLE

Time complexity: $O(n^2 \log n^2)$
Space complexity: $O(n^2)$

Solution 1: Binary Search Accepted

Related Problems: LC 378/719

Search m, 0 < m < 1
Such that there are exact K pairs of (i,j) that A[i] / A[j] <= m

Create a virtual matrix to speed up the search process

Time complexity: $O(\log(\max)n^2)$ TLE
$O(\log(\max)n)$
Space complexity: O(1)

smallest

1/2	1/3	1/5
-	2/3	2/5
-	-	3/5

A = [1, 2, 3, 5], K = 3

def: M[i][j] = A[i] / A[j]
0 <= i <= n - 1, i < j < n - 1
M[i][j] < M[i][j - 1]
M[i][j] > M[i][j - 1]

if M[i][j] > m => M[i + 1][j] >= m

l	r	m	A[i] / A[j] <= m	total
0	1	0.5	1/2, 1/3, 1/5, 2/5	4 > K
0	0.5	0.25	1/5	1 < K
0.25	0.5	0.375	1/3, 1/5	2 < K
0.375	0.5	0.4375	1/3, 1/5, 2/5	3 = K

http://zxi.mytechroad.com/blog/

Figure 3.14: K-th Smallest Prime Fraction

```
    def kthSmallestPrimeFraction(self, A, K):
        l = 0.0
        r = 1.0
        n = len(A)

        while l < r:
            mid = (l + r) / 2
            max_f = 0.0
            total = 0
            j = 1
            for i in range(n - 1):
                while j < n and A[i] > A[j] * mid:
                    j += 1
                if j == n:
                    break
                total += n-j
                f = A[i] / A[j]
                if f > max_f:
                    p,q,max_f = i,j,f
            if total == K:
                return [A[p], A[q]]
            elif total > K:
                r = mid
            else:
                l = mid

        return []
```

3.6 Dynamic Programming

Normally the problem of DP is the problem of **finding maximum or minumum solutions**, that is to say, to find optimal solution in operation research, for example, longest increasing sub-sequence, minimum edit distance. To find the extremum, the core is **exhaustive method**, a little different from pure exhaustic method, DP related questions should be optimized using DP table, after the dp table, the dynamic transition function and optimal sub structure is the key to the answer. So the most difficult part is to write down the **dynamic transition function**. The process to write down DTF is as follows:
status -¿ define the meaning of dp array or matrix -¿ know what to choose -¿ know what is the base case

3.7 Two Pointers

3.8 HashTable

#1 TwoSum I, a basic form of these problems, given an array and a integer **target**. Return the index of these 2 numbers.

A simple brute force solution:

```
def twoSum(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: List[int]
    """
    l = len(nums)
    for i in range(l):
        for j in range(i+1, l):
            if nums[j] == target - nums[i]:
                return [i, j]

    return [-1, -1]
```

We could use hashtable here, since the search complexity of hashtable is O(n), then it'll be easy to find the number and return the index. And the key in the hashtable is the element nums[i] and value is the index i. It is better to use collections.defaultdict(), which will be faster.

```
    def twoSum(self, nums, target):
        hashtable = {}
        for i in range(len(nums)):
            hashtable[nums[i]] = i

        for i in range(len(nums)):
            if target - nums[i] in hashtable and hashtable[target - nums[i]] != i:
                return [i, hashtable[target - nums[i]]]

        return []
```

The problem could also be done by using two pointers technique, the prerequisite is that the the given array is sorted.

Chapter 4

Tensorflow

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special contents, but the length of words should match the language.