3. Longest Substring Without Repeating Characters

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring
```

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

```
def lengthOfLongestSubstring(self, s: str) -> int:
        char = [0] * 128
        left = right = 0
        res = 0
        while right < len(s):
            r = s[right]
            char[ord(r)] += 1
            while char[ord(r)] > 1:
                l = s[left]
                char[ord(l)] -= 1
                left += 1
            res = max(res, right-left+1)
            right +=1
        return res
```

5. Longest Palindromic Substring

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

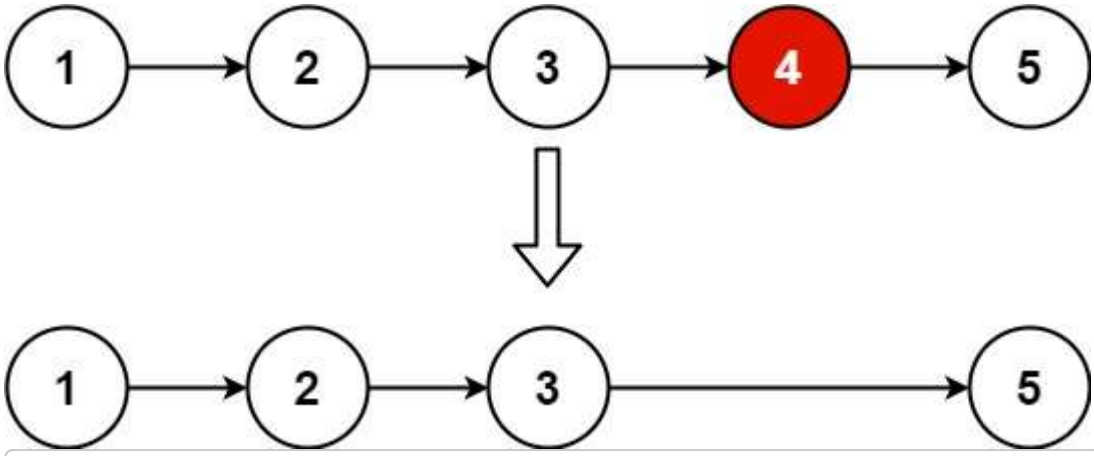
len 1 - if longest pallindrome is of odd length

```
class Solution:
    def checkPalindrome(self,s, l, r):
        while(l>=0 and r<len(s) and s[l]==s[r]):
            l-=1;
            r+=1;
        return(r-l-1)
    def longestPalindrome(self, s: str) -> str:
        start = 0
        end = 0
        res = ""
        for i in range(len(s)):
            len1 = self.checkPalindrome(s,i,i) # lcp is of odd length
            len2 = self.checkPalindrome(s,i,i+1) # lcp is of even length
            leng = max(len1,len2)
            if(leng>end-start):
                start = i-(leng-1)//2
                end = i+leng//2
                res = s[start:end+1]
        return(res)
```

19. Remove Nth Node From End of List

Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:




```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Example 2:

```
Input: head = [1], n = 1
Output: []
```

Example 3:

```
Input: head = [1,2], n = 1
Output: [1]
```

Constraints:

- The number of nodes in the list is `sz` .
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

Follow up: Could you do this in one pass?

```
def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        temp = ListNode(0)
        temp.next = head
        slw = fast = temp
        for i in range(n):
            fast = fast.next
        while fast.next:
            slw = slw.next
            fast = fast.next
        slw.next = slw.next.next
        return temp.next
```

---

35. Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`
- `nums` contains **distinct** values sorted in **ascending** order.
- `-10^4 <= target <= 10^4`

---

3. Binary search

```
def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            if nums[mid] == target:
                return mid
            if target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        return left
```

---

46. Permutations

Given an array `nums` of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

Example 1:

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Example 2:

```
Input: nums = [0,1]
Output: [[0,1],[1,0]]
```

Example 3:

```
Input: nums = [1]
Output: [[1]]
```

Constraints:

- `1 <= nums.length <= 6`
- `-10 <= nums[i] <= 10`
- All the integers of `nums` are **unique**.

---

index based backtrack, swap

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        def backtrack(first=0):
            if first == n:
                res.append(nums[:])
            for i in range(first, n):
                nums[first], nums[i] = nums[i], nums[first]
                backtrack(first+1)
                nums[first], nums[i] = nums[i], nums[first]
        n = len(nums)
        res = []
        backtrack()
        return res
```

---

70. Climbing Stairs

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 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
```

Constraints:

- `1 <= n <= 45`

Recursion(time limit):

```
def climbStairs(self, n: int) -> int:
        def climb(i=0):
            if i > n:
                return 0
            if i == n:
                return 1
            return climb(i+1)+climb(i+2)
        return climb(0)
```

Recursion with memo:

```
def climbStairs(self, n: int) -> int:
        memo = [0] * (n+1)
        def climb(i=0):
            if i > n:
                return 0
            if i == n:
                return 1
            if memo[i] > 0:
                return memo[i]
            memo[i] = climb(i+1)+climb(i+2)
            return memo[i]
        return climb()
```

DP

```
def climbStairs(self, n: int) -> int:
        dp = [0]*(n+1)
        if n == 1:
            return 1
        dp[1] = 1
        dp[2]  = 2
        for i in range(3, n+1):
            dp[i] = dp[i-1] + dp[i-2]
        return dp[n]
```

---

77. Combinations

Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers out of the range* `[1, n]`.

You may return the answer in **any order**.

Example 1:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Example 2:

```
Input: n = 1, k = 1
Output: [[1]]
```

Constraints:

- `1 <= n <= 20`
- `1 <= k <= n`

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        def backtrack(first=1, curr=[]):
            if len(curr) == k:
                res.append(curr[:])
                return # not required. as it will be exited in the next line
            for i in range(first, n+1):
                curr.append(i)
                backtrack(i+1, curr)
                curr.pop() # imp to go one step back and append other value
        res = []
        backtrack()
        return res
```

---

116. Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

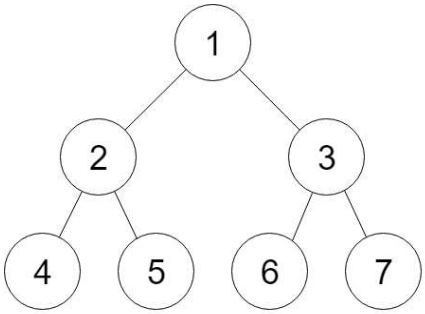


Figure A

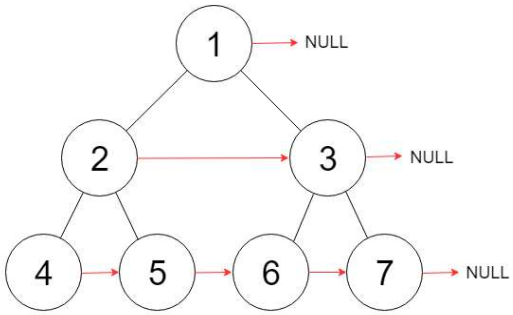


Figure B

```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should popul
```

Example 2:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2^12 - 1]`.
- `-1000 <= Node.val <= 1000`

Follow-up:

- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

```
def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        if not root:
            return root
        q = [root]
        while q:
            size = len(q)
            for i in range(size):
                node = q.pop(0)
                if i < size-1:
                    node.next = q[0]
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
        return root
```

167. Two Sum II - Input Array Is Sorted

Given a **1-indexed** array of integers `numbers` that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where `1 <= index1 < index2 <= numbers.length`.

Return *the indices of the two numbers,* `index1` *and* `index2`*,* ***added by one*** *as an integer array* `[index1, index2]` *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

Example 1:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2]
```

Example 2:

```
Input: numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3]
```

Example 3:

```
Input: numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1,
```

Constraints:

- `2 <= numbers.length <= 3 * 10^4`
- `-1000 <= numbers[i] <= 1000`
- `numbers` is sorted in **non-decreasing order**.
- `-1000 <= target <= 1000`
- The tests are generated such that there is **exactly one solution**.

2P

```
def twoSum(self, numbers: List[int], target: int) -> List[int]:
        left , right = 0, len(numbers)-1
        while left < right:
            if numbers[left] + numbers[right] == target:
                return [left+1, right+1]
            elif numbers[left] + numbers[right] < target:
                left += 1
            else:
                right -= 1
        return [-1, -1]
```

---

189. Rotate Array

Given an array, rotate the array to the right by `k` steps, where `k` is non-negative.

Example 1:

```
Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Constraints:

- `1 <= nums.length <= 10^5`
- `-2^31 <= nums[i] <= 2^31 - 1`
- `0 <= k <= 10^5`

Follow up:

- Try to come up with as many solutions as you can. There are at least **three** different ways to solve this problem.
- Could you do it in-place with `O(1)` extra space?

---

5. 2P

```
def reverse(self, nums, left, right):
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left, right = left+1, right-1
    def rotate(self, nums: List[int], k: int) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        n = len(nums)
        k %= n
        self.reverse(nums, 0, n-1)
        self.reverse(nums, 0, k-1)
        self.reverse(nums, k, n-1)
```

198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money =
Total amount you can rob = 2 + 9 + 1 = 12.
```

Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

Recurrsion with Memo

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        memo = {}
        n = len(nums)
        def rec(i=0):
            if i >= n:
                return 0
            if i in memo:
                return memo[i]
            #if you choose the subsequent next current earning will be discarded => o
nly rec(i+1)
            #if you choose the alternate next current earning will be coumpunde => on
ly rec(i+2)+nums[i]
            memo[i] = max(rec(i+1), rec(i+2)+nums[i])
            return memo[i]
        return rec()
```

200. Number of Islands

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'` s (land) and `'0'` s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is `'0'` or `'1'`.

BFS

```
def numIslands(self, grid: List[List[str]]) -> int:
        R, C = len(grid), len(grid[0])
        res = 0
        for i in range(R):
            for j in range(C):
                if grid[i][j] is "0":
                    continue
                res += 1
                Q = [(i,j)]
                grid[i][j] = "0"
                while Q: #continue untill all connected 1's are marked 0
                    r, c = Q.pop(0)
                    for nexti,nextj in [(r-1,c), (r+1,c), (r,c-1), (r,c+1)]:
                        if 0<=nexti<R and 0<nextj<C and grid[nexti][nextj] == "1":
                            grid[nexti][nextj] = "0"
                            Q.append((nexti,nextj))
        return res
```

DFS

```
def numIslands(self, grid: List[List[str]]) -> int:
        R, C = len(grid), len(grid[0])
        stack = []
        islands = 0

        def dfs(r, c):
            stack.append((r,c))
            while stack:
                i,j = stack.pop()
                grid[i][j] = '0'
                for nexti, nextj in ((i+1, j), (i, j+1), (i-1, j), (i, j-1)):
                    if 0 <= nexti < R and 0 <= nextj < C and grid[nexti][nextj] ==
'1':
                        stack.append((nexti, nextj))
        for i in range(R):
            for j in range(C):
                if grid[i][j] == '1':
                    islands += 1
                    dfs(i, j)
        return islands
```

---

210. Course Schedule II

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [a_i, b_i]` indicates that you **must** take course `b_i` first if you want to take course `a_i`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have f
```

Example 2:

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have f
So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].
```

Example 3:

```
Input: numCourses = 1, prerequisites = []
Output: [0]
```

Constraints:

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= numCourses * (numCourses - 1)`
- `prerequisites[i].length == 2`
- `0 <= a_i, b_i < numCourses`
- `a_i != b_i`
- All the pairs `[a_i, b_i]` are **distinct**.

Topological sort

(Orrdering vertices of graph such that for every edge u->v, v should only appear after u)

---

1. Pick any node. If not present in visited add to visited.

2. Go to its child node. add to visited if not present already

3. continue 1 and 2 untill a node is found which has no child or all children are visited

4. add the above node to the stack

5. Go back to its parent and repeat.

6. print the stack in reverse order.

(Since we are adding nodes to the stack when all its children are visited, Topology rule is satisfied)

Comlexity: classical dfs, so time comlexity is O(E+V)

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[in
t]:
        from collections import defaultdict
        graph = defaultdict(set)
        visited = [0] * numCourses # 0:not yet visited, 1:visiting, 2: visited
        self.ans, self.cycle = [], 0

        for i,j in prerequisites:
            graph[i].add(j)  # {1:(0), 2:(0), 3:(1,2)} key:course, val: pre

        def dfs(start):
            if self.cycle == 1: # cycle
                return
            if visited[start] == 1: # if the current node is already getting visited
mark cycle
                self.cycle = 1
            if visited[start] == 0:
                visited[start] = 1 # started visiting
                for i in graph[start]:
                    dfs(i)
                visited[start] = 2 # all childs are visited, so mark visited
                self.ans.append(start) # since all child are visited add to stack

        for i in range(numCourses):
            if self.cycle:
                break
            if visited[i] == 0:
                dfs(i)
        return [] if self.cycle else self.ans
```

231. Power of Two

Given an integer `n`, return *`true` if it is a power of two. Otherwise, return `false`*.

An integer `n` is a power of two, if there exists an integer `x` such that `n == 2^x`.

Example 1:

```
Input: n = 1
Output: true
Explanation: 2^0 = 1
```

Example 2:

```
Input: n = 16
Output: true
Explanation: 2^4 = 16
```

Example 3:

```
Input: n = 3
Output: false
```

Constraints:

- $-2^{31} <= n <= 2^{31} - 1$

Follow up: Could you solve it without loops/recursion?

---

x and -x (2's complement- switch all bits + 1) have just one bit in common - the rightmost 1-bit. That means that x & (-x) would keep that rightmost 1-bit and set all the other bits to 0.

power of two contains just one 1-bit

Hence a number is a power of two if x & (-x) == x

```
def isPowerOfTwo(self, n: int) -> bool:
        if not n:
            return False
        return n & (-n) == n
```

---

253. Meeting Rooms II

Given an array of meeting time intervals `intervals` where `intervals[i] = [start_i, end_i]`, return *the minimum number of conference rooms required*.

Example 1:

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
```

Example 2:

```
Input: intervals = [[7,10],[2,4]]
Output: 1
```

Constraints:

- `1 <= intervals.length <= 10^4`
- `0 <= start_i < end_i <= 10^6`

By default heapq- min heap

```
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x:x[0])
        free_rooms = []
        heapq.heappush(free_rooms, intervals[0][1])
        for i in intervals[1:]:
            if free_rooms[0] <= i[0]:
                heapq.heappop(free_rooms)
            heapq.heappush(free_rooms, i[1])
        return len(free_rooms)
```

---

278. First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

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

You are given an API `bool isBadVersion(version)` which returns whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Example 1:

```
Input: n = 5, bad = 4
Output: 4
Explanation:
call isBadVersion(3) -> false
call isBadVersion(5) -> true
call isBadVersion(4) -> true
Then 4 is the first bad version.
```

Example 2:

```
Input: n = 1, bad = 1
Output: 1
```

Constraints:

- `1 <= bad <= n <= 2^31 - 1`

---

2. Binary search

```
def firstBadVersion(self, n: int) -> int:
        left, right = 0, n
        while left < right:
            mid = (left + right)//2
            if isBadVersion(mid):
                right = mid
            else:
                left = mid+1
        return left
```

---

283. Move Zeroes

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

```
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Example 2:

```
Input: nums = [0]
Output: [0]
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-2^31 <= nums[i] <= 2^31 - 1`

Follow up: Could you minimize the total number of operations done?

---

6. 2P

```
def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        left, right = 0, len(nums)-1

        while left < right:
            if nums[left] == 0:
                nums.pop(left)
                nums.append(0)
                right -= 1
            else:
                left += 1
```

---

344. Reverse String

Write a function that reverses a string. The input string is given as an array of characters `s`.

You must do this by modifying the input array in-place (https://en.wikipedia.org/wiki/In-place_algorithm) with `O(1)` extra memory.

Example 1:

```
Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
```

Example 2:

```
Input: s = ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]
```

Constraints:

- `1 <= s.length <= 10^5`
- `s[i]` is a printable ascii character (https://en.wikipedia.org/wiki/ASCII#Printable_characters).

8

```
def reverseString(self, s: List[str]) -> None:
        left, right = 0, len(s)-1
        while left<right:
            s[left], s[right] = s[right], s[left]
            left += 1
            right -= 1
```

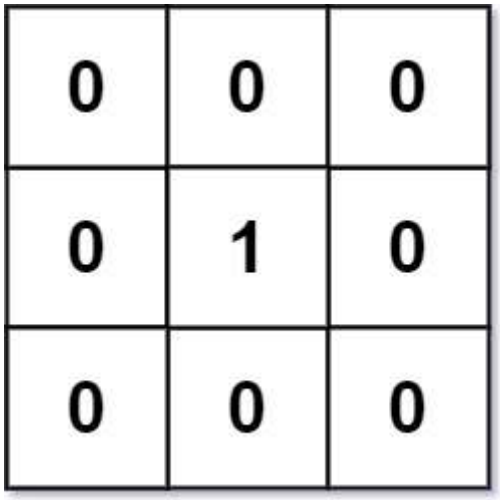
---

542. 01 Matrix

Given an `m x n` binary matrix `mat`, return *the distance of the nearest* `0` *for each cell.*

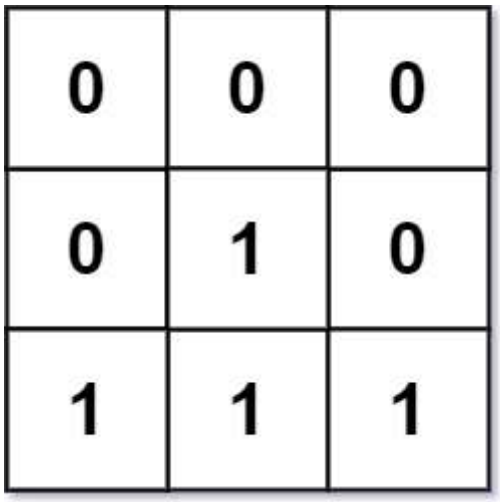
The distance between two adjacent cells is `1`.

Example 1:




```
Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]
```

Example 2:




```
Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]
```

Constraints:

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 10^4`
- `1 <= m * n <= 10^4`
- `mat[i][j]` is either `0` or `1`.
- There is at least one `0` in `mat`.

BFS from 0. Update cells

```
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        R, C = len(mat), len(mat[0])
        Q = []
        for i in range(R):
            for j in range(C):
                if mat[i][j] == 0:
                    Q.append((i,j))
                else:
                    mat[i][j] = -1
        while Q:
            r, c = Q.pop(0)
            for i, j in ((r+1, c), (r-1, c), (r, c+1), (r, c-1)):
                if 0<=i<R and 0<=j<C and mat[i][j] == -1:
                    mat[i][j] = mat[r][c] + 1 # increase 1 from the "from cell"
                    Q.append((i,j)) # add  to Q to propogate distances
        return mat
```

---

557. Reverse Words in a String III

Given a string `s`, reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

Example 1:

```
Input: s = "Let's take LeetCode contest"
Output: "s'teL ekat edoCteeL tsetnoc"
```

Example 2:

```
Input: s = "God Ding"
Output: "doG gniD"
```

Constraints:

- `1 <= s.length <= 5 * 10^4`
- `s` contains printable **ASCII** characters.
- `s` does not contain any leading or trailing spaces.
- There is **at least one** word in `s`.
- All the words in `s` are separated by a single space.

9.

```
def reverseWords(self, s: str) -> str:
        return " ".join([i[::-1] for i in s.split(' ')])
```

---

567. Permutation in String

Given two strings `s1` and `s2`, return `true` *if* `s2` *contains a permutation of* `s1`, *or* `false` *otherwise.*

In other words, return `true` if one of `s1`'s permutations is the substring of `s2`.

Example 1:

```
Input: s1 = "ab", s2 = "eidbaooo"
Output: true
Explanation: s2 contains one permutation of s1 ("ba").
```

Example 2:

```
Input: s1 = "ab", s2 = "eidboaoo"
Output: false
```

Constraints:

- `1 <= s1.length, s2.length <= 10^4`
- `s1` and `s2` consist of lowercase English letters.

```
def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
            return False
        s1_chars = [0] * 26
        s2_chars = [0] * 26
        for i in s1:
            s1_chars[ord(i)-97] += 1
        for i in range(len(s2)-len(s1)+1): # +1 because diff index shall be included
in start
            for j in range(i, i+len(s1)):
                s2_chars[ord(s2[j])-97] += 1
            if s1_chars == s2_chars:
                return True
            else:
                s2_chars = [0] * 26
        return False
```

---

617. Merge Two Binary Trees

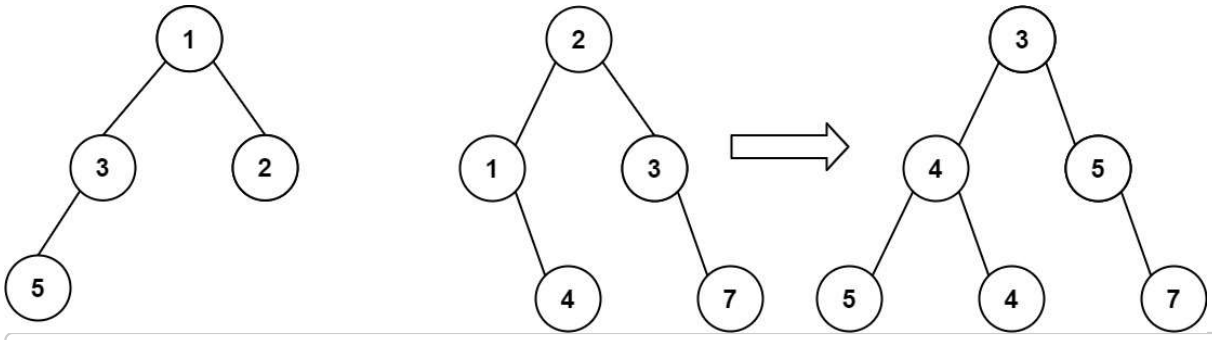
You are given two binary trees `root1` and `root2`.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

Note: The merging process must start from the root nodes of both trees.

Example 1:




```
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
Output: [3,4,5,5,4,null,7]
```

Example 2:

```
Input: root1 = [1], root2 = [1,2]
Output: [2,2]
```

Constraints:

- The number of nodes in both trees is in the range `[0, 2000]` .
- `-10^4 <= Node.val <= 10^4`

```
def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optiona
l[TreeNode]:
        if not root1: return root2
        if not root2: return root1
        root1.val += root2.val
        root1.left = self.mergeTrees(root1.left, root2.left)
        root1.right = self.mergeTrees(root1.right, root2.right)
        return root1
```

---

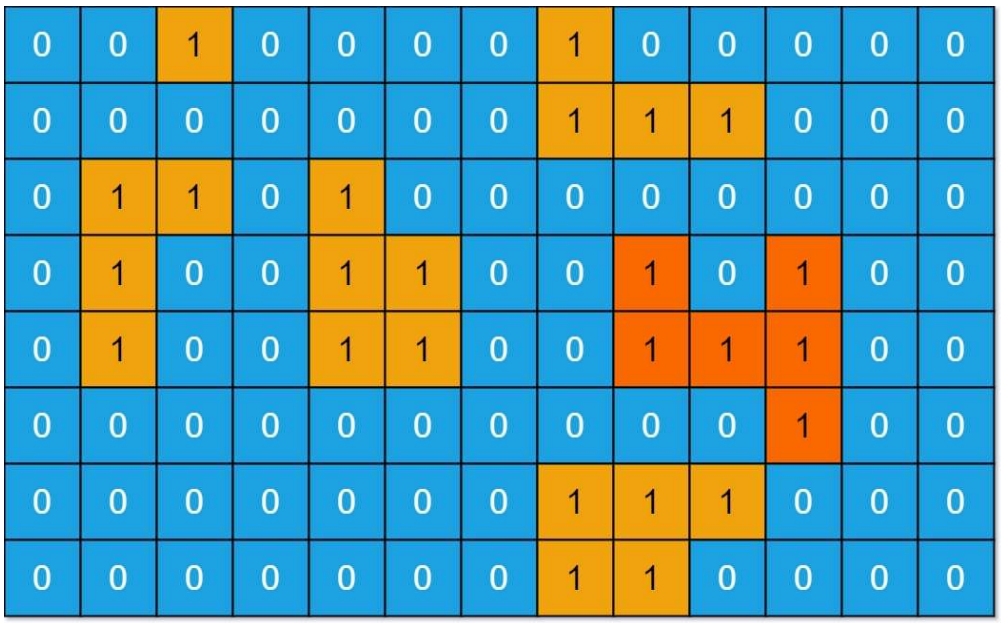
695. Max Area of Island

You are given an `m x n` binary matrix `grid` . An island is a group of `1` 's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value `1` in the island.

Return *the maximum **area** of an island in* `grid` . If there is no island, return `0` .

Example 1:




```
Input: grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0
Output: 6
Explanation: The answer is not 11, because the island must be connected 4-directionally
```

Example 2:

```
Input: grid = [[0,0,0,0,0,0,0,0]]
Output: 0
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 50`
- `grid[i][j]` is either `0` or `1`.

```
def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        seen = set()
        R, C = len(grid), len(grid[0])
        def area(r, c):
            if not(0<=r<R and 0<=c<C and (r,c) not in seen and grid[r][c]):
                return 0
            seen.add((r,c))
            return (1+area(r-1,c)+area(r,c-1)+area(r+1,c)+area(r,c+1))
        return max(area(i,j) for i in range(R) for j in range(C))
```

---

733. Flood Fill

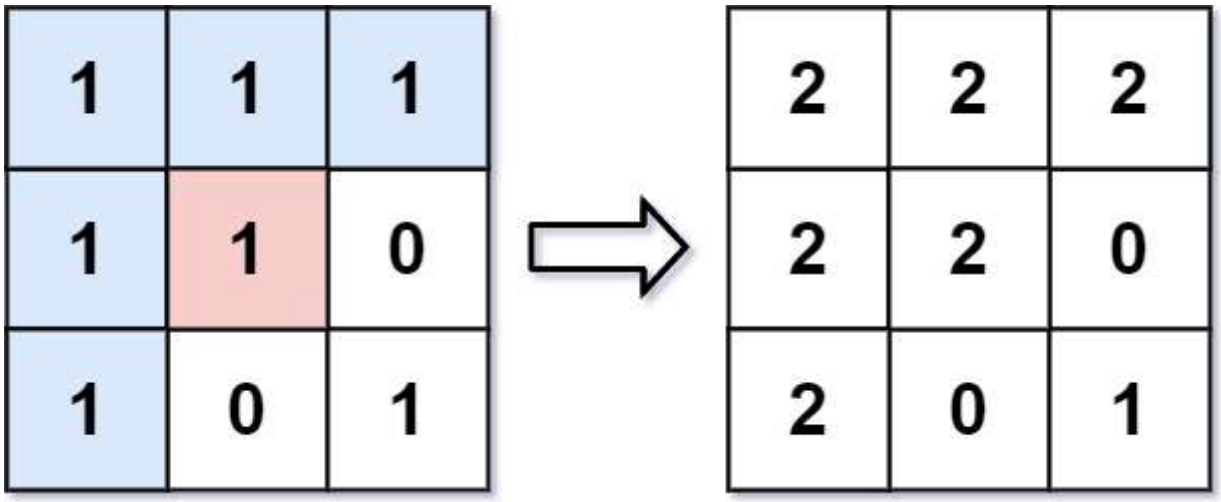
An image is represented by an `m x n` integer grid `image` where `image[i][j]` represents the pixel value of the image.

You are also given three integers `sr`, `sc`, and `newColor`. You should perform a **flood fill** on the image starting from the pixel `image[sr][sc]`.

To perform a **flood fill**, consider the starting pixel, plus any pixels connected **4-directionally** to the starting pixel of the same color as the starting pixel, plus any pixels connected **4-directionally** to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with `newColor`.

Return *the modified image after performing the flood fill.*

Example 1:




```
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation: From the center of the image with position (sr, sc) = (1, 1) (i.e., the re
Note the bottom corner is not colored 2, because it is not 4-directionally connected to
```

Example 2:

```
Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, newColor = 2
Output: [[2,2,2],[2,2,2]]
```

Constraints:

- `m == image.length`
- `n == image[i].length`

- `1 <= m, n <= 50`
- `0 <= image[i][j], newColor < 2^16`
- `0 <= sr < m`
- `0 <= sc < n`

DFS- Iterative

```
class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, newColor: int) -> L
ist[List[int]]:
        R, C = len(image), len(image[0])
        color = image[sr][sc]
        if color == newColor:
            return image
        stack = [(sr,sc)]
        while stack:
            i, j = stack.pop()
            if image[i][j] == color:
                image[i][j] = newColor
                if i>=1: stack.append((i-1, j))
                if j>=1: stack.append((i, j-1))
                if i<R-1: stack.append((i+1, j))
                if j<C-1: stack.append((i, j+1))
        return image
```

704. Binary Search

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^4 < nums[i], target < 10^4`
- All the integers in `nums` are **unique**.
- `nums` is sorted in ascending order.

---

1. Binary Search

```
def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums)-1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid
            elif nums[mid] > target:
                right = mid - 1
            else:
                left = mid + 1
        return -1
```

---

876. Middle of the Linked List

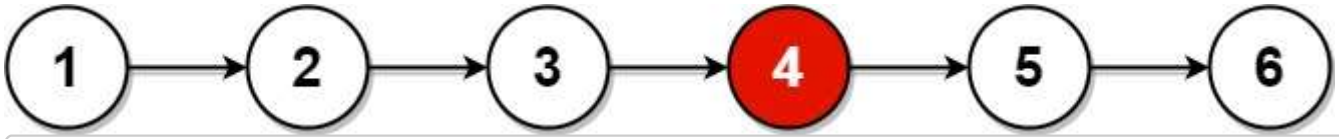
Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Example 1:


```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2:




```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the sec
```

Constraints:

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`

```
def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slw = head
        fst = head
        while fst and fst.next:
            slw = slw.next
            fst =  fst.next.next
        return slw
```

977. Squares of a Sorted Array

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of* ***the squares of each number*** *sorted in non-decreasing order.*

Example 1:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].
```

Example 2:

```
Input: nums = [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`
- `nums` is sorted in **non-decreasing** order.

Follow up: Squaring each element and sorting the new array is very trivial, could you find an `O(n)` solution using a different approach?

---

4. BS

```
def sortedSquares(self, nums: List[int]) -> List[int]:
        n = len(nums)
        squares = [0]*n
        left, right = 0, n-1
        for i in range(n-1, -1, -1):
            if abs(nums[right]) > abs(nums[left]):
                square = nums[right]
                right-=1
            else:
                square = nums[left]
                left+=1
            squares[i] = square ** 2
        return squares
```

---

994. Rotting Oranges

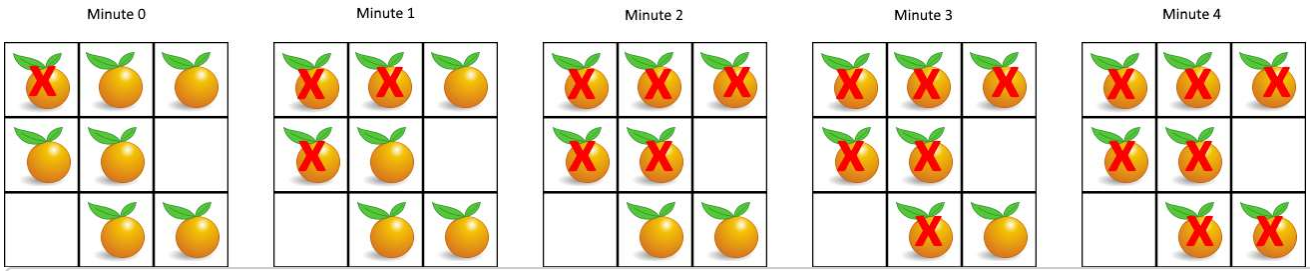
You are given an `m x n grid` where each cell can have one of three values:

- `0` representing an empty cell,
- `1` representing a fresh orange, or
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* `-1`.

Example 1:




```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

Example 2:

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, be
```

Example 3:

```
Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 10`
- `grid[i][j]` is `0`, `1`, or `2`.

BFS

```
def orangesRotting(self, grid: List[List[int]]) -> int:
        R, C = len(grid), len(grid[0])
        Q = []
        fresh_oranges = 0
        days = 0
        for i in range(R):
            for j in range(C):
                if grid[i][j] == 2:
                    Q.append((i, j))
                elif grid[i][j] == 1:
                    fresh_oranges += 1
        Q.append((-1, -1)) # to keep track of end of day
        while Q:
            r, c = Q.pop(0)
            if r == -1: # if EOD
                days += 1
                if Q: # if items left add -1 to track next EOD
                    Q.append((-1,-1))
            else:
                for i, j in ((r+1,c), (r-1,c), (r,c+1), (r,c-1)):
                    if 0<=i<R and 0<=j<C and grid[i][j]==1:
                        grid[i][j] = 2
                        fresh_oranges -= 1
                        Q.append((i,j))
        return days-1 if not fresh_oranges else -1
```

1268. Search Suggestions System

You are given an array of strings `products` and a string `searchWord`.

Design a system that suggests at most three product names from `products` after each character of `searchWord` is typed. Suggested products should have common prefix with `searchWord`. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return *a list of lists of the suggested products after each character of* `searchWord` *is typed*.

Example 1:

```
Input: products = ["mobile","mouse","moneypot","monitor","mousepad"], searchWord = "mou
Output: [
["mobile","moneypot","monitor"],
["mobile","moneypot","monitor"],
["mouse","mousepad"],
["mouse","mousepad"],
["mouse","mousepad"]
]
Explanation: products sorted lexicographically = ["mobile","moneypot","monitor","mouse"
After typing m and mo all products match and we show user ["mobile","moneypot","monitor
After typing mou, mous and mouse the system suggests ["mouse","mousepad"]
```

Example 2:

```
Input: products = ["havana"], searchWord = "havana"
Output: [["havana"],["havana"],["havana"],["havana"],["havana"],["havana"]]
```

Example 3:

```
Input: products = ["bags","baggage","banner","box","cloths"], searchWord = "bags"
Output: [["baggage","bags","banner"],["baggage","bags","banner"],["baggage","bags"],["b
```

Constraints:

- `1 <= products.length <= 1000`
- `1 <= products[i].length <= 3000`
- `1 <= sum(products[i].length) <= 2 * 10^4`
- All the strings of `products` are **unique**.
- `products[i]` consists of lowercase English letters.
- `1 <= searchWord.length <= 1000`
- `searchWord` consists of lowercase English letters.

```
class TrieNode:
    def __init__(self):
        self.children = dict()
        self.words = []

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            node.words.append(word)
            node.words.sort()
            while len(node.words) > 3:
                node.words.pop()

    def search(self, word):
        res = []
        node = self.root
        for char in word:
            if char not in node.children:
                break
            node = node.children[char]
            res.append(node.words[:])
        l_remain = len(word) - len(res)
        for _ in range(l_remain):
            res.append([])
        return res

class Solution:
    def suggestedProducts(self, products: List[str], searchWord: str) -> List[List[st
r]]:
        trie = Trie()
        for prod in products:
            trie.insert(prod)
        return trie.search(searchWord)
```