8. String to Integer (atoi)

Implement the `myAtoi(string s)` function, which converts a string to a 32-bit signed integer.

The algorithm for `myAtoi(string s)` is as follows:

1. **Whitespace**: Ignore any leading whitespace (`" "`).
2. **Signedness**: Determine the sign by checking if the next character is `'-'` or `'+'`, assuming positivity if neither present.
3. **Conversion**: Read the integer by skipping leading zeros until a non-digit character is encountered or the end of the string is reached. If no digits were read, then the result is 0.
4. **Rounding**: If the integer is out of the 32-bit signed integer range $[-2^{31}, 2^{31} - 1]$, then round the integer to remain in the range. Specifically, integers less than -2^{31} should be rounded to -2^{31}, and integers greater than $2^{31} - 1$ should be rounded to $2^{31} - 1$.

Return the integer as the final result.

Example 1:

Input: s = "42"

Output: 42

Explanation:

```
The underlined characters are what is read in and the caret is the current reader pos
Step 1: "42" (no characters read because there is no leading whitespace)
         ^
Step 2: "42" (no characters read because there is neither a '-' nor '+')
         ^
Step 3: "42" ("42" is read in)
           ^
```

Example 2:

Input: s = " -042"

Output: -42

Explanation:

```
Step 1: "___-042" (leading whitespace is read and ignored)
            ^
Step 2: "   -042" ('-' is read, so the result should be negative)
             ^
Step 3: "   -042" ("042" is read in, leading zeros ignored in the result)
               ^
```

Example 3:

Input: s = "1337c0d3"

Output: 1337

Explanation:

```
Step 1: "1337c0d3" (no characters read because there is no leading whitespace)
         ^
Step 2: "1337c0d3" (no characters read because there is neither a '-' nor '+')
         ^
Step 3: "1337c0d3" ("1337" is read in; reading stops because the next character is a
             ^
```

Example 4:

Input: s = "0-1"

Output: 0

Explanation:

```
Step 1: "0-1" (no characters read because there is no leading whitespace)
         ^
Step 2: "0-1" (no characters read because there is neither a '-' nor '+')
         ^
Step 3: "0-1" ("0" is read in; reading stops because the next character is a non-digi
          ^
```

Example 5:

Input: s = "words and 987"

Output: 0

Explanation:

Reading stops at the first non-digit character 'w'.

Constraints:

- `0 <= s.length <= 200`
- `s` consists of English letters (lower-case and upper-case), digits (`0-9`), `' '`, `'+'`, `'-'`, and `'.'`.

```
class Solution:
    def helper(self,s,i,ans):
        if i < len(s) and s[i] in "0123456789":
            return self.helper(s,i+1,ans+s[i])
        return ans

    def myAtoi(self, s: str) -> int:
        s = s.strip()
        if not s:return 0
        ans = 1
        if s[0] == "-":
            ans = -1
            s = s[1:]
        elif s[0] == "+":
            s = s[1:]
        temp = self.helper(s,0,"")
        if not temp:return 0
        ans = int(temp) * ans
        if ans < -2**31 :
            ans = -2**31
        elif ans > (2**31) - 1:
            ans = (2**31) - 1
        return ans
```

17. Letter Combinations of a Phone Number

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

Example 3:

```
Input: digits = "2"
Output: ["a","b","c"]
```

Constraints:

- `0 <= digits.length <= 4`
- `digits[i]` is a digit in the range `['2', '9']`.

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:return []
        my_dict = {
            "2": "abc",
            "3": "def",
            "4": "ghi",
            "5": "jkl",
            "6": "mno",
            "7": "pqrs",
            "8": "tuv",
            "9": "wxyz"
        }
        res = []
        def backtrack(ind,temp):
            if ind >= len(digits):
                res.append(temp)
                return
            digit = digits[ind]
            for j in range(len(my_dict[digit])):
                backtrack(ind+1,temp+my_dict[digit][j])

        backtrack(0,"")
        return res
```

---

22. Generate Parentheses

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

Constraints:

- `1 <= n <= 8`

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        res = []
        def recursion(open,close,temp):
            if len(temp)== n*2:
                res.append(temp)
                return
            if open < n:
                recursion(open+1,close,temp+"(")
            if close < open:
                recursion(open,close+1,temp+")")
        recursion(1,0,"(")
        return res
```

37. Sudoku Solver

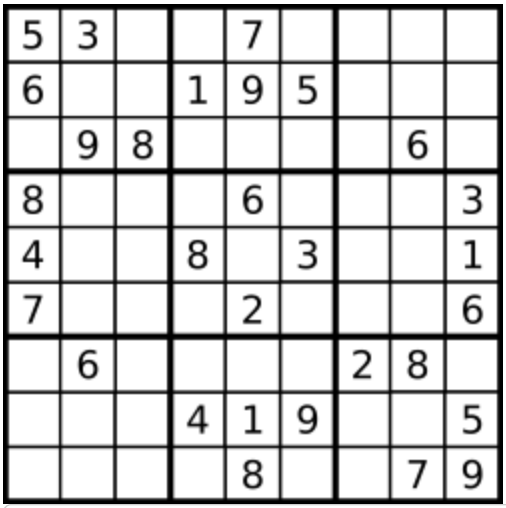
Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

Example 1:



Input: board = [["5","3",".",".","7",".",".",".",".",".],["6",".",".","1","9","5",".","
Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],
Explanation: The input board is shown above and the only valid solution is shown belo

Constraints:

- `board.length == 9`
- `board[i].length == 9`
- `board[i][j]` is a digit or `'.'`.
- It is **guaranteed** that the input board has only one solution.

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        def is_valid(board,row,col,ch):
            for i in range(0,9):
                if board[row][i] == ch or board[i][col] == ch:return False
                if board[3*(row//3)+i//3][3*(col//3)+i%3] == ch:return False
            return True

        def solve(board):
            for i in range(9):
                for j in range(9):
                    if board[i][j] == ".":
                        for k in range(1,10):
                            if is_valid(board,i,j,str(k)):
                                board[i][j] = str(k)
                                if solve(board):
                                    return True
                                board[i][j] = "."
                        return False
            return True
        solve(board)
```

39. Combination Sum

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

```
Input: candidates = [2], target = 1
Output: []
```

Constraints:

- `1 <= candidates.length <= 30`
- `2 <= candidates[i] <= 40`
- All elements of `candidates` are **distinct**.
- `1 <= target <= 40`

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[in
t]]:
        res = []
        def recursion(ind,temp,summ):
            if summ == target:
                res.append(temp[:])
                return
            if ind >= len(candidates) or summ > target:
                return
            temp.append(candidates[ind])
            recursion(ind,temp,summ+candidates[ind])
            temp.pop()
            recursion(ind+1,temp,summ)
        recursion(0,[],0)
        return res
```

---

40. Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target` .

Each number in `candidates` may only be used **once** in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8
Output:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]
```

Example 2:

```
Input: candidates = [2,5,2,1,2], target = 5
Output:
[
[1,2,2],
[5]
]
```

Constraints:

- `1 <= candidates.length <= 100`
- `1 <= candidates[i] <= 50`
- `1 <= target <= 30`

```
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[in
t]]:
        res= []
        candidates.sort()
        def recursion(start,summ,subset):
            if summ == target:
                res.append(subset[:])
                return
            if summ > target or start >= len(candidates):
                return

            for ind in range(start,len(candidates)):
                if ind > start and candidates[ind] == candidates[ind-1]:
                    continue
                subset.append(candidates[ind])
                recursion(ind+1,summ+candidates[ind],subset)
                subset.pop()
        recursion(0,0,[])
        return res
```

---

50. Pow(x, n)

Implement pow(x, n) (http://www.cplusplus.com/reference/valarray/pow/), which calculates `x` raised to the power `n` (i.e., x^n).

Example 1:

```
Input: x = 2.00000, n = 10
Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3
Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

Constraints:

- -100.0 < x < 100.0
- -2^{31} <= n <= $2^{31}-1$
- n is an integer.
- Either x is not zero or n > 0 .
- -10^4 <= x^n <= 10^4

If n is even: xn=xn/2·xn/2

If n is odd: xn=xLn/2⌋·xLn/2⌋·x

```
class Solution:
    def helper(self,x,n):
        if n == 0:return 1
        half = self.helper(x,n//2)
        if n%2 == 0:
            return half * half
        else:
            return half*half*x

    def myPow(self, x: float, n: int) -> float:
        ans  = self.helper(x,n if n > 0 else n*-1)
        return ans if n>0 else 1/ans
```

---

78. Subsets

Given an integer array nums of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

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

Example 2:

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

Constraints:

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

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []
        def recursion(idx,temp):
            if idx >= len(nums):
                res.append(temp.copy())
                return
            temp.append(nums[idx])
            recursion(idx+1,temp)
            temp.pop()
            recursion(idx+1,temp)
        recursion(0,[])
        return res
```

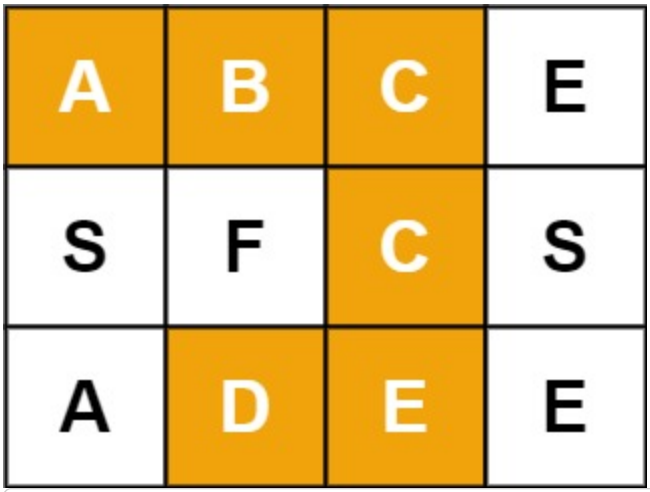
---

79. Word Search

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid*.

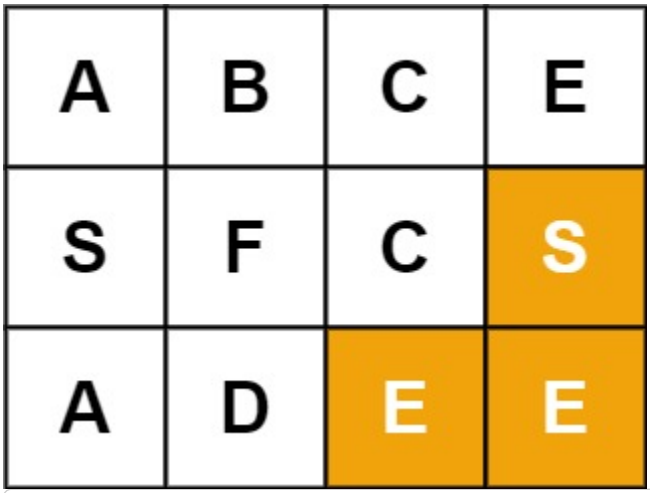
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



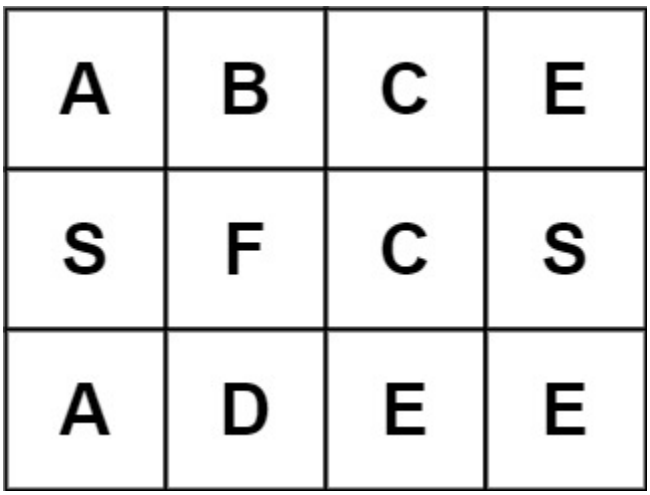

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCE
Output: true
```

Example 2:




```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true
```

Example 3:



```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false
```

Constraints:

- `m == board.length`
- `n = board[i].length`
- `1 <= m, n <= 6`
- `1 <= word.length <= 15`
- `board` and `word` consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger `board`?

Get better space coplexity by using a temp = board[i][j] board[i][j] = "#" and then again asign the temp board = [i][j]

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        m = len(board)
        n = len(board[0])
        def recursion(i,j,index):
            if index >= len(word):return True
            if not (0 <= i < m) or not (0<= j < n) or (i,j) in vis or board[i][j] !
= word[index]:return False
            vis.add((i,j))
            if recursion(i+1,j,index+1) or recursion(i,j+1,index+1) or recursion
(i,j-1,index+1) or recursion(i-1,j,index+1):
                return True
            vis.remove((i, j))
            return False
        for i in range(m):
            for j in range(n):
                vis = set()
                if board[i][j] == word[0] :
                    if recursion(i,j,0):
                        return True
        return False
```

---

90. Subsets II

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

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

Example 2:

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

Constraints:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        res = []
        nums.sort()
        def recursion(ind,subset):
            res.append(subset[:])
            if ind == len(nums):
                return
            for i in range(ind,len(nums)):
                if i > ind and nums[i] == nums[i-1]:
                    continue
                subset.append(nums[i])
                recursion(i+1,subset)
                subset.pop()
        recursion(0,[])
        return res
```

131. Palindrome Partitioning

Given a string `s`, partition `s` such that every substring of the partition is a **palindrome**. Return *all possible palindrome partitioning of* `s`.

Example 1:

```
Input: s = "aab"
Output: [["a","a","b"],["aa","b"]]
```

Example 2:

```
Input: s = "a"
Output: [["a"]]
```

Constraints:

- `1 <= s.length <= 16`
- `s` contains only lowercase English letters.

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        #https://leetcode.com/problems/palindrome-partitioning/
        res = []
        def backtrack(start,temp):
            if start == len(s):
                res.append(temp[:])
                return
            for end in range(start+1,len(s)+1):
                if s[start:end] == s[start:end][::-1]:
                    backtrack(end,temp+ [s[start:end]])

        backtrack(0,[])
        return res
```

216. Combination Sum III

Find all valid combinations of `k` numbers that sum up to `n` such that the following conditions are true:

- Only numbers `1` through `9` are used.
- Each number is used **at most once**.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

Example 1:

```
Input: k = 3, n = 7
Output: [[1,2,4]]
Explanation:
1 + 2 + 4 = 7
There are no other valid combinations.
```

Example 2:

```
Input: k = 3, n = 9
Output: [[1,2,6],[1,3,5],[2,3,4]]
Explanation:
1 + 2 + 6 = 9
1 + 3 + 5 = 9
2 + 3 + 4 = 9
There are no other valid combinations.
```

Example 3:

```
Input: k = 4, n = 1
Output: []
Explanation: There are no valid combinations.
Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4
```

Constraints:

- `2 <= k <= 9`
- `1 <= n <= 60`

```
use style 2 as that is preferred [for loop combination good for maintaining order a
nd duplicates ]

class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        res = []
        def recursion(ind,summ,subset):
            if n == summ and len(subset) == k:
                res.append(subset[:])
                return
            if len(subset) >= k or summ > n or ind > 9:
                return

            subset.append(ind)
            recursion(ind+1,summ+ind,subset)
            subset.pop()
            recursion(ind+1,summ,subset)
        recursion(1,0,[])
        return res


from typing import List

class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        result = []

        def backtrack(start: int, current_sum: int, path: List[int]):
            # Base case: valid combination
            if len(path) == k and current_sum == n:
                result.append(path[:])
                return
            # Pruning: stop if invalid path
            if len(path) > k or current_sum > n:
                return

            for i in range(start, 10):  # Only digits 1 through 9
                path.append(i)
                backtrack(i + 1, current_sum + i, path)
                path.pop()

        backtrack(1, 0, [])
        return result
```

282. Expression Add Operators

Given a string `num` that contains only digits and an integer `target`, return ***all possibilities*** *to insert the binary operators* `'+'`, `'-'`, *and/or* `'*'` *between the digits of* `num` *so that the resultant expression evaluates to the* `target` *value.*

Note that operands in the returned expressions **should not** contain leading zeros.

Example 1:

```
Input: num = "123", target = 6
Output: ["1*2*3","1+2+3"]
Explanation: Both "1*2*3" and "1+2+3" evaluate to 6.
```

Example 2:

```
Input: num = "232", target = 8
Output: ["2*3+2","2+3*2"]
Explanation: Both "2*3+2" and "2+3*2" evaluate to 8.
```

Example 3:

```
Input: num = "3456237490", target = 9191
Output: []
Explanation: There are no expressions that can be created from "3456237490" to evalua
```

Constraints:

- `1 <= num.length <= 10`
- `num` consists of only digits.
- `-2^31 <= target <= 2^31 - 1`

```
from typing import List

class Solution:
    def addOperators(self, num: str, target: int) -> List[str]:
        res = []

        def backtrack(index: int, path: str, eval_val: int, prev_num: int):
            if index == len(num):
                if eval_val == target:
                    res.append(path)
                return

            for i in range(index, len(num)):
                # Skip numbers with leading zeros
                if i != index and num[index] == '0':
                    break

                curr_str = num[index:i + 1]
                curr_num = int(curr_str)

                if index == 0:
                    # First number, start expression
                    backtrack(i + 1, curr_str, curr_num, curr_num)
                else:
                    # '+'
                    backtrack(i + 1, path + "+" + curr_str, eval_val + curr_num, cu
rr_num)
                    # '-'
                    backtrack(i + 1, path + "-" + curr_str, eval_val - curr_num, -c
urr_num)
                    # '*'
                    backtrack(i + 1, path + "*" + curr_str,
                              eval_val - prev_num + (prev_num * curr_num),
                              prev_num * curr_num)

        backtrack(0, "", 0, 0)
        return res
```

1922. Count Good Numbers

A digit string is **good** if the digits **(0-indexed)** at **even** indices are **even** and the digits at **odd** indices are **prime** (`2` , `3` , `5` , or `7`).

- For example, `"2582"` is good because the digits (`2` and `8`) at even positions are even and the

digits (`5` and `2`) at odd positions are prime. However, `"3245"` is **not** good because `3` is at an even index but is not even.

Given an integer `n` , return *the **total** number of good digit strings of length* `n` . Since the answer may be large, **return it modulo** $10^9 + 7$.

A **digit string** is a string consisting of digits `0` through `9` that may contain leading zeros.

Example 1:

```
Input: n = 1
Output: 5
Explanation: The good numbers of length 1 are "0", "2", "4", "6", "8".
```

Example 2:

```
Input: n = 4
Output: 400
```

Example 3:

```
Input: n = 50
Output: 564908303
```

Constraints:

- $1 <= n <= 10^{15}$

---

https://leetcode.com/problems/count-good-numbers/solutions/6645467/beats-super-easy-beginners-java-c-c-python-javascript-dart/ (https://leetcode.com/problems/count-good-numbers/solutions/6645467/beats-super-easy-beginners-java-c-c-python-javascript-dart/)

```
class Solution:
    def countGoodNumbers(self, n: int) -> int:
        mod = 10**9 + 7
        even = (n + 1) // 2
        odd = n // 2
        return (pow(5, even, mod) * pow(4, odd, mod)) % mod
```
