面试题 08.12. 八皇后

```
class Solution:
    def __init__(self):
        self.result = []
    def solveNQueens(self, n: int) -> List[List[str]]:
        board = [["."] * n for i in range(n)]
        self.backtrack(0, board, n)
        return self.result

    def backtrack(self, row, board, n):
        if row == n:
            snapshot = []
            for i in range(n):
                snapshot.append("".join(board[i]))
            self.result.append(snapshot)
        for col in range(n):
            if self.isOK(board, n, row, col):
                board[row][col] = "Q"
                self.backtrack(row+1, board, n)
                board[row][col] = "."
        return

    def isOK(self, board, n, row, col):
        #检查列判断
        for i in range(n):
            if board[i][col] == "Q":
                return False
        #检查又对角线
        i = row - 1
        j = col + 1
        while i >= 0 and j < n:
            if board[i][j] == "Q":
                return False
            i -= 1
            j += 1
        #检查左对角线
        i = row - 1
        j = col - 1
        while i >= 0 and j >= 0:
            if board[i][j] == "Q":
                return False
            i -= 1
            j -= 1
        return True
```

37. 解数独

```
class Solution:
    def __init__(self):
        self.solved = False
        self.rows = [[None] * 10 for i in range(9)]
        self.cols = [[None] * 10 for i in range(9)]
        self.blocks = [[[None] * 10 for i in range(3)] for i in range(3)]
```

```
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        for i in range(9):
            for j in range(9):
                if board[i][j] != '.':
                    num = int(board[i][j])
                    self.rows[i][num] = True
                    self.cols[j][num] = True
                    self.blocks[i//3][j//3][num] = True
        self.backtrack(0,0,board)
        return self.solved

    def backtrack(self, row, col, board):
        if row == 9:
            self.solved = True
            return
        if board[row][col] !=".":
            nextRow = row
            nextCol = col + 1
            if col == 8:
                nextRow = row + 1
                nextCol = 0
            self.backtrack(nextRow, nextCol, board)
            if self.solved:
                return
        else:
            for num in range(1,10):
                if not self.rows[row][num] and not self.cols[col][num] and not self.blocks[row//3][col//
3][num]:
                    board[row][col] = str(num)
                    self.rows[row][num] = True
                    self.cols[col][num] = True
                    self.blocks[row//3][col//3][num] = True
                    nextRow = row
                    nextCol = col + 1
                    if col == 8:
                        nextRow = row + 1
                        nextCol = 0
                    self.backtrack(nextRow, nextCol, board)
                    if self.solved:
                        return
                    board[row][col] = "."
                    self.rows[row][num] = False
                    self.cols[col][num] = False
                    self.blocks[row//3][col//3][num] = False
```

17. 电话号码的字母组合

```
class Solution:
    def __init__(self):
        self.result = []
    def letterCombinations(self, digits: str) -> List[str]:
```

```
        if len(digits) == 0:
            return self.result
        mappings = [None] * 10
        mappings[2] = "abc"
        mappings[3] = "def"
        mappings[4] = "ghi"
        mappings[5] = "jkl"
        mappings[6] = "mno"
        mappings[7] = "pqrs"
        mappings[8] = "tuv"
        mappings[9] = "wxyz"
        path = [None] * len(digits)
        self.backtrack(mappings, digits, 0, path)
        return self.result

    def backtrack(self, mappings, digits, k, path):
        if k == len(digits):
            self.result.append("".join(path))
            return
        mapping = mappings[int(digits[k])]
        for i in range(len(mapping)):
            path[k] = mapping[i]
            self.backtrack(mappings, digits, k+1, path)
```

77. 组合

```
class Solution:
    def __init__(self):
        self.result = []
    def combine(self, n: int, k: int) -> List[List[int]]:
        self.backtrack(n, k, 1, [])
        return self.result

    def backtrack(self, n, k, step, path):
        if len(path) == k:
            self.result.append(path)
            return
        if step == n + 1:
            return
        self.backtrack(n, k, step+1, path)
        path.append(step)
        self.backtrack(n,k,step+1, path)
        path.pop()
```

78. 子集

```
class Solution:
    def __init__(self):
        self.result = []
    def subsets(self, nums: List[int]) -> List[List[int]]:
        self.backtrack(nums, 0, [])
        return self.result

    def backtrack(self, nums, k, path):
```

```
        if k == len(nums):
            self.result.append(path[:])
            return
        self.backtrack(nums, k+1,path)
        path.append(nums[k])
        self.backtrack(nums, k+1,path)
        path.pop()
```

90. 子集 II

```
class Solution:
    def __init__(self):
        self.result = []
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        hm = dict()
        for i in range(len(nums)):
            count = 1
            if nums[i] in hm:
                count += hm[nums[i]]
            hm[nums[i]] = count
        n = len(hm)
        uniqueNum = [None] * n
        counts = [None] * n
        k = 0
        for i in range(len(nums)):
            if nums[i] in hm:
                uniqueNum[k] = nums[i]
                counts[k] = hm[nums[i]]
                k +=1
                hm.pop(nums[i])
        self.backtrack(uniqueNum, counts, 0, [])
        return self.result

    def backtrack(self, uniqueNum, counts, k, path):
        if  k == len(uniqueNum):
            self.result.append(path[:])
            return
        for count in range(counts[k]+1):
            for i in range(count):
                path.append(uniqueNum[k])
            self.backtrack(uniqueNum, counts, k+1, path)
            for i in range(count):
                path.pop()
```

46. 全排列

```
class Solution:
    def __init__(self):
        self.result = []
    def permute(self, nums: List[int]) -> List[List[int]]:
        self.backtrack(nums, 0, [])
        return self.result

    def backtrack(self, nums, k, path):
        if k == len(nums):
```

```
            self.result.append(path[:])
            return
        for i in range(len(nums)):
            if nums[i] in path:
                continue
            path.append(nums[i])
            self.backtrack(nums, k+1,path)
            path.pop()
```

47. 全排列 II

```
class Solution:
    def __init__(self):
        self.result = []
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        hm = dict()
        for i in range(len(nums)):
            count = 1
            if nums[i] in hm:
                count += hm[nums[i]]
            hm[nums[i]] = count
        n = len(hm)
        uniqueNum = [None] * n
        counts = [None] * n
        k = 0
        for i in range(len(nums)):
            if nums[i] in hm:
                uniqueNum[k] = nums[i]
                counts[k] = hm[nums[i]]
                k +=1
                hm.pop(nums[i])
        self.backtrack(uniqueNum, counts, 0, [], len(nums))
        return self.result

    def backtrack(self, uniqueNum, counts, k, path,n):
        if  k == n:
            self.result.append(path[:])
            return
        for i in range(len(uniqueNum)):
            if counts[i] == 0:
                continue
            path.append(uniqueNum[i])
            counts[i] -= 1
            self.backtrack(uniqueNum, counts, k+1, path, n)
            path.pop()
            counts[i] += 1
```

39. 组合总和

```
class Solution:
    def __init__(self):
        self.result = []
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        self.backtrack(candidates,0,target,[])
        return self.result
```

```
    def backtrack(self, candidates, k, left, path):
        if left == 0:
            self.result.append(path[:])
            return
        if  k == len(candidates):
            return
        for i in range(left //candidates[k] +1):
            for j in range(i):
                path.append(candidates[k])
            self.backtrack(candidates, k+1, left - candidates[k] * i, path)
            for j in range(i):
                path.pop()
```

40. 组合总和 II

```
class Solution:
    def __init__(self):
        self.result = []
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        hashTable = dict()
        for i in range(len(candidates)):
            if candidates[i] not in hashTable:
                hashTable[candidates[i]] = 1
            else:
                hashTable[candidates[i]] += 1
        nums = []
        counts = []
        for i in range(len(candidates)):
            if candidates[i] in hashTable:
                nums.append(candidates[i])
                counts.append(hashTable[candidates[i]])
                hashTable.pop(candidates[i])

        self.backtrack(nums, counts, 0, target, [])
        return self.result

    def backtrack(self, nums, counts, k, left, path):
        if left == 0:
            self.result.append(path[:])
            return
        if k == len(nums) or left < 0:
            return
        for count in range(counts[k] +1):
            for i in range(count):
                path.append(nums[k])
            self.backtrack(nums, counts, k+1, left - count*nums[k], path)
            for i in range(count):
                path.pop()
```

216. 组合总和 III

```
class Solution:
    def __init__(self):
        self.result = []
```

```
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        self.backtrack(k,n,1,0,[])
        return self.result

    def backtrack(self, k, n, step, total, path):
        if total == n and len(path) == k:
            self.result.append(path[:])
            return
        if total > n or len(path) > k or step > 9:
            return
        self.backtrack(k,n,step+1,total,path)
        path.append(step)
        self.backtrack(k,n,step+1,total+step,path)
        path.pop()
```

131. 分割回文串

```
class Solution:
    def __init__(self):
        self.result = []
    def partition(self, s: str) -> List[List[str]]:
        self.backtrack(s, 0, [])
        return self.result

    def backtrack(self,s, k, path):
        if k == len(s):
            self.result.append(path[:])
            return
        for end in range(k, len(s)):
            if self.ispalindrome(s, k, end):
                path.append(s[k:end+1])
                self.backtrack(s, end+1,path)
                path.pop()

    def ispalindrome(self,s, p,r):
        i = p
        j = r
        while i <= j:
            if s[i] !=s[j]:
                return False
            i += 1
            j -= 1
        return True
```

93. 复原 IP 地址

```
class Solution:
    def __init__(self):
        self.result = []
    def restoreIpAddresses(self, s: str) -> List[str]:
        self.backtrack(s, 0,0, [])
        return self.result

    def backtrack(self, s, k, step, path):
        if step == 4 and k == len(s):
```

```
            sb = [str(item) for item in path]
            self.result.append(".".join(sb))
            return
        if step > 4:
            return
        if k == len(s):
            return
        val = 0
        if k < len(s):
            val = val * 10 + int(s[k])
            path.append(val)
            self.backtrack(s, k+1, step+1,path)
            path.pop()
        if s[k] == "0":
            return
        if k + 1< len(s):
            val = val * 10 + int(s[k+1])
            path.append(val)
            self.backtrack(s, k+2, step+1,path)
            path.pop()
        if k + 2< len(s):
            val = val * 10 + int(s[k+2])
            if val <= 255:
                path.append(val)
                self.backtrack(s, k+3, step+1,path)
                path.pop()
```

22. 括号生成

```
class Solution:
    def __init__(self):
        self.result = []
    def generateParenthesis(self, n: int) -> List[str]:
        path = [None] * 2 * n
        self.backtrack(n, 0, 0,0,path)
        return self.result

    def backtrack(self,n,leftUsed,rightUesd, k,path):
        if k == 2*n:
            self.result.append("".join(path))
            return
        if leftUsed > rightUesd:
            path[k]=")"
            self.backtrack(n,leftUsed,rightUesd+1, k+1, path)
        if leftUsed < n:
            path[k]="("
            self.backtrack(n,leftUsed+1,rightUesd, k+1, path)
```