1. Largest K elements

```
from heapq import *


def kLargestElements(arr, k):
    heap = []
    for i in arr:
        if len(heap) == k:
            heappushpop(heap, i)
        else:
            heappush(heap, i)
    return heap
```

37. Sudoku Solver

We will use 27 sets to record all rows, columns and small cubes status. TC is O(1), SC is O(9*9*3)

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        sets = [set() for i in range(27)]
        count = 9 * 9
        for i in range(9):
            for j in range(9):
                if board[i][j] != '.':
                    sets[i].add(board[i][j])
                    sets[9 + j].add(board[i][j])
                    sets[18 + i // 3 * 3 + j // 3].add(board[i][j])
                    count -= 1
        self.fill(board, sets, 0, 0, count)

    def fill(self, board, sets, start_i, start_j, rest):
        if rest == 0:
            return True
        i, j = start_i, start_j
        while i < 9:
            while j < 9:
                if board[i][j] == '.':
                    for k in range(1, 10):
                        ch = str(k)
                        if ch not in sets[i] and ch not in sets[9 + j] and ch not in sets[18 + i
// 3 * 3 + j // 3]:
```

```
                    board[i][j] = ch
                    sets[i].add(ch)
                    sets[9 + j].add(ch)
                    sets[18 + i // 3 * 3 + j // 3].add(ch)
                    if self.fill(board, sets, i, j, rest-1):
                        return True
                    sets[i].remove(ch)
                    sets[9 + j].remove(ch)
                    sets[18 + i // 3 * 3 + j // 3].remove(ch)
                    board[i][j] = '.'
                return False
            j += 1
        j = 0
        i += 1
    return True
```

5. Longest Palindromic Substring

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        max_length = 0
        max_substring = ''
        for i, _ in enumerate(s):
            sub = self.helper(i, i + 1, s)
            if len(sub) > max_length:
                max_length = len(sub)
                max_substring = sub
            sub = self.helper(i - 1, i + 1, s)
            if len(sub) > max_length:
                max_length = len(sub)
                max_substring = sub
        return max_substring

    def helper(self, l, r, s):
        while l >= 0 and r < len(s):
            if s[l] == s[r]:
                l -= 1
                r += 1
            else:
                break

        return s[l + 1: r]
```

47. Permutations II

We will use iteration to insert each num to each previous arrays. When we encounter same element, we will break to prevent duplication. TC is O(n ^ 2)

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        ans = [[]]
        for num in nums:
            cur = []
            for l in ans:
                for i in range(len(l) + 1):
                    cur.append(l[:i] + [num] + l[i:])
                    if i < len(l) and l[i] == num:
                        break
            ans = cur
        return ans
```

47. Permutations II

```
from collections import Counter
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        res = []
        counter = Counter(nums)
        def backTrack(cur):
            if len(cur) == len(nums):
                res.append(cur[:])
                print(res)
                return
            for k, v in counter.items():
                if v <= 0:
                    continue
                cur.append(k)
                counter[k] -= 1
                backTrack(cur)
                cur.pop()
                counter[k] += 1
        backTrack([])
                        return res
```