```python
# 1- Valid anagram:

from collections import Counter

def are_anagrams(s1, s2):
    if len(s1) != len(s2):
        return False
    return Counter(s1) == Counter(s2)


def are_anagrams(s1, s2):
    if len(s1) != len(s2):
        return False
    return sorted(s1) == sorted(s2)

def first_and_last(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            start = i
            while i+1 < len(arr) and arr[i+1] == target:
                i += 1
            return [start, i]
    return [-1, -1]
def find_start(arr, target):
    if arr[0] == target:
        return 0
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid] == target and arr[mid-1] < target:
            return mid
        elif arr[mid] < target:
            left = mid+1
        else:
            right = mid-1
    return -1
def find_end(arr, target):
    if arr[-1] == target:
        return len(arr)-1
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid] == target and arr[mid+1] > target:
            return mid
        elif arr[mid] > target:
            right = mid-1
        else:
            left = mid+1
    return -1
def first_and_last(arr, target):
    if len(arr) == 0 or arr[0] > target or arr[-1] < target:
        return [-1, -1]
    start = find_start(arr, target)
    end = find_end(arr, target)
    return [start, end]


# 3- Kth largest element:

def kth_largest(arr, k):
    for i in range(k - 1):
        arr.remove(max(arr))
    return max(arr)


def kth_largest(arr, k):
    n = len(arr)
    arr.sort()
    return arr[n - k]


import heapq


def kth_largest(arr, k):
    arr = [-elem for elem in arr]
    heapq.heapify(arr)
    for i in range(k - 1):
        heapq.heappop(arr)
    return -heapq.heappop(arr)


# 4- Symmetric tree:

def are_symmetric(root1, root2):
    if root1 is None and root2 is None:
        return True
    elif ((root1 is None) != (root2 is None)) or root1.val != root2.val:
        return False
    else:
        return are_symmetric(root1.left, root2.right) and are_symmetric(root1.right, root2.left)


def is_symmetric(root):
    if root is None:
        return True
    return are_symmetric(root.left, root.right)

# 5- Generate parentheses:

def generate(n):
    def rec(n, diff, comb, combs):
        if diff < 0 or diff > n:
            return
        elif n == 0:
            if diff == 0:
                combs.append(''.join(comb))
        else:
            comb.append('(')
            rec(n-1, diff+1, comb, combs)
            comb.pop()
            comb.append(')')
            rec(n-1, diff-1, comb, combs)
            comb.pop()
    combs = []
    rec(2*n, 0, [], combs)
    return combs
```

```

# 6- Gas station:

def can_traverse(gas, cost, start):
    n = len(gas)
    remaining = 0
    i = start
    started = False
    while i != start or not started:
        started = True
        remaining += gas[i] - cost[i]
        if remaining < 0:
            return False
        i = (i+1)%n
    return True


def gas_station(gas, cost):
    for i in range(len(gas)):
        if can_traverse(gas, cost, i):
            return i
    return -1
def gas_station(gas, cost):
    remaining = 0
    prev_remaining = 0
    candidate = 0
    for i in range(len(gas)):
        remaining += gas[i] - cost[i]
        if remaining < 0:
            candidate = i+1
            prev_remaining += remaining
            remaining = 0
    if candidate == len(gas) or remaining+prev_remaining < 0:
        return -1
    else:
        return candidate

    # 7- Course schedule:

 def dfs(graph, vertex, path, order, visited):
        path.add(vertex)
        for neighbor in graph[vertex]:
            if neighbor in path:
                return False
            if neighbor not in visited:
                visited.add(neighbor)
                if not dfs(graph, neighbor, path, order, visited):
                    return False
        path.remove(vertex)
        order.append(vertex)
        return True

def course_schedule(n, prerequisites):
    graph = [[] for i in range(n)]
    for pre in prerequisites:
        graph[pre[1]].append(pre[0])
    visited = set()
    path = set()
    order = []
    for course in range(n):
        if course not in visited:
            visited.add(course)
            if not dfs(graph, course, path, order, visited):
                return False
    return True


from collections import deque
def course_schedule(n, prerequisites):
    graph = [[] for i in range(n)]
    indegree = [0 for i in range(n)]
    for pre in prerequisites:
        graph[pre[1]].append(pre[0])
        indegree[pre[0]] += 1
    order = []
    queue = deque([i for i in range(n) if indegree[i] == 0])
    while queue:
        vertex = queue.popleft()
```

```python
        order.append(vertex)
        for neighbor in graph[vertex]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
    return len(order) == n

# 8- Kth permutation:

import itertools

def kth_permutation(n, k):
    permutations = list(itertools.permutations(range(1, n+1)))
    return ''.join(map(str, permutations[k-1]))


def kth_permutation(n, k):
    permutation = []
    unused = list(range(1, n+1))
    fact = [1]*(n+1)
    for i in range(1, n+1):
        fact[i] = i*fact[i-1]
    k -= 1
    while n > 0:
        part_length = fact[n]//n
        i = k//part_length
        permutation.append(unused[i])
        unused.pop(i)
        n -= 1
        k %= part_length
    return ''.join(map(str, permutation))


# 9- Minimum window substring:

def contains_all(freq1, freq2):
    for ch in freq2:
        if freq1[ch] < freq2[ch]:
            return False
    return True
```

```python


def min_window(s, t):
    n, m = len(s), len(t)
    if m > n or m == 0:
        return ""
    freqt = Counter(t)
    shortest = " "*(n+1)
    for length in range(1, n+1):
        for i in range(n-length+1):
            sub = s[i:i+length]
            freqs = Counter(sub)
            if contains_all(freqs, freqt) and length < len(shortest):
                shortest = sub
    return shortest if len(shortest) <= n else ""
def min_window(s, t):
    n, m = len(s), len(t)
    if m > n or t == "":
        return ""
    freqt = Counter(t)
    start, end = 0, n+1
    for length in range(1, n+1):
        freqs = Counter()
        satisfied = 0
        for ch in s[:length]:
            freqs[ch] += 1
            if ch in freqt and freqs[ch] == freqt[ch]:
                satisfied += 1
        if satisfied == len(freqt) and length < end-start:
            start, end = 0, length
        for i in range(1, n-length+1):
            freqs[s[i+length-1]] += 1
            if s[i+length-1] in freqt and freqs[s[i+length-1]] == freqt[s[i+length-1]]:
                satisfied += 1
            if s[i-1] in freqt and freqs[s[i-1]] == freqt[s[i-1]]:
                satisfied -= 1
            freqs[s[i-1]] -= 1
            if satisfied == len(freqt) and length < end-start:
                start, end = i, i+length
    return s[start:end] if end-start <= n else ""
def min_window(s, t):
    n, m = len(s), len(t)
    if m > n or t == "":
        return ""
    freqt = Counter(t)
    start, end = 0, n
    satisfied = 0
    freqs = Counter()
    left = 0
    for right in range(n):
        freqs[s[right]] += 1
        if s[right] in freqt and freqs[s[right]] == freqt[s[right]]:
            satisfied += 1
        if satisfied == len(freqt):
            while s[left] not in freqt or freqs[s[left]] > freqt[s[left]]:
                freqs[s[left]] -= 1
                left += 1
            if right-left+1 < end-start+1:
                start, end = left, right
    return s[start:end+1] if end-start+1 <= n else ""

# 10- Largest rectangle in histogram:

def largest_rectangle(heights):
    max_area = 0
    for i in range(len(heights)):
        left = i
        while left-1 >= 0 and heights[left-1] >= heights[i]:
            left -= 1
        right = i
        while right+1 < len(heights) and heights[right+1] >= heights[i]:
            right += 1
        max_area = max(max_area, heights[i]*(right-left+1))
    return max_area


def rec(heights, low, high):
    if low > high:
        return 0
    elif low == high:
        return heights[low]
    else:
        minh = min(heights[low:high + 1])
        pos_min = heights.index(minh, low, high + 1)
        from_left = rec(heights, low, pos_min - 1)
        from_right = rec(heights, pos_min + 1, high)
        return max(from_left, from_right, minh * (high - low + 1))


def largest_rectangle(heights):
    return rec(heights, 0, len(heights) - 1)
def largest_rectangle(heights):
    heights = [-1]+heights+[-1]
    from_left = [0]*len(heights)
    stack = [0]
    for i in range(1, len(heights)-1):
        while heights[stack[-1]] >= heights[i]:
            stack.pop()
        from_left[i] = stack[-1]
        stack.append(i)
    from_right = [0]*len(heights)
    stack = [len(heights)-1]
    for i in range(1, len(heights)-1)[::-1]:
        while heights[stack[-1]] >= heights[i]:
            stack.pop()
        from_right[i] = stack[-1]
        stack.append(i)
    max_area = 0
    for i in range(1, len(heights)-1):
        max_area = max(max_area, heights[i]*(
from_right[i]-from_left[i]-1))
    return max_area
def largest_rectangle(heights):
    heights = [-1]+heights+[-1]
    max_area = 0
    stack = [(0, -1)]
    for i in range(1, len(heights)):
        start = i
        while stack[-1][1] > heights[i]:
            top_index, top_height = stack.pop()
            max_area = max(max_area, top_height*(i-top_index))
            start = top_index
        stack.append((start, heights[i]))
    return max_area






```