```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
```

```
40      left, right = 0, len(arr)-1
41      while left <= right:
42          mid = (left+right)//2
43          if arr[mid] == target and arr[mid+1] > target
    :
44              return mid
45          elif arr[mid] > target:
46              right = mid-1
47          else:
48              left = mid+1
49      return -1
50  def first_and_last(arr, target):
51      if len(arr) == 0 or arr[0] > target or arr[-1] <
    target:
52          return [-1, -1]
53      start = find_start(arr, target)
54      end = find_end(arr, target)
55      return [start, end]
56
57
58  # 3- Kth largest element:
59
60  def kth_largest(arr, k):
61      for i in range(k - 1):
62          arr.remove(max(arr))
63      return max(arr)
64
65
66  def kth_largest(arr, k):
67      n = len(arr)
68      arr.sort()
69      return arr[n - k]
70
71
72  import heapq
73
74
75  def kth_largest(arr, k):
76      arr = [-elem for elem in arr]
77      heapq.heapify(arr)
78      for i in range(k - 1):
```

```
 79         heapq.heappop(arr)
 80     return -heapq.heappop(arr)
 81
 82
 83 # 4- Symmetric tree:
 84
 85 def are_symmetric(root1, root2):
 86     if root1 is None and root2 is None:
 87         return True
 88     elif ((root1 is None) != (root2 is None)) or
    root1.val != root2.val:
 89         return False
 90     else:
 91         return are_symmetric(root1.left, root2.right
    ) and are_symmetric(root1.right, root2.left)
 92
 93
 94 def is_symmetric(root):
 95     if root is None:
 96         return True
 97     return are_symmetric(root.left, root.right)
 98
 99 # 5- Generate parentheses:
100
101 def generate(n):
102     def rec(n, diff, comb, combs):
103         if diff < 0 or diff > n:
104             return
105         elif n == 0:
106             if diff == 0:
107                 combs.append(''.join(comb))
108         else:
109             comb.append('(')
110             rec(n-1, diff+1, comb, combs)
111             comb.pop()
112             comb.append(')')
113             rec(n-1, diff-1, comb, combs)
114             comb.pop()
115     combs = []
116     rec(2*n, 0, [], combs)
117     return combs
```

```
118
119 # 6- Gas station:
120
121 def can_traverse(gas, cost, start):
122     n = len(gas)
123     remaining = 0
124     i = start
125     started = False
126     while i != start or not started:
127         started = True
128         remaining += gas[i] - cost[i]
129         if remaining < 0:
130             return False
131         i = (i+1)%n
132     return True
133
134
135 def gas_station(gas, cost):
136     for i in range(len(gas)):
137         if can_traverse(gas, cost, i):
138             return i
139     return -1
140 def gas_station(gas, cost):
141     remaining = 0
142     prev_remaining = 0
143     candidate = 0
144     for i in range(len(gas)):
145         remaining += gas[i] - cost[i]
146         if remaining < 0:
147             candidate = i+1
148             prev_remaining += remaining
149             remaining = 0
150     if candidate == len(gas) or remaining+
    prev_remaining < 0:
151         return -1
152     else:
153         return candidate
154
155     # 7- Course schedule:
156
157  def dfs(graph, vertex, path, order, visited):
```

```
158         path.add(vertex)
159         for neighbor in graph[vertex]:
160             if neighbor in path:
161                 return False
162             if neighbor not in visited:
163                 visited.add(neighbor)
164                 if not dfs(graph, neighbor, path,
    order, visited):
165                     return False
166         path.remove(vertex)
167         order.append(vertex)
168         return True
169 
170 def course_schedule(n, prerequisites):
171     graph = [[] for i in range(n)]
172     for pre in prerequisites:
173         graph[pre[1]].append(pre[0])
174     visited = set()
175     path = set()
176     order = []
177     for course in range(n):
178         if course not in visited:
179             visited.add(course)
180             if not dfs(graph, course, path, order,
    visited):
181                 return False
182     return True
183 
184 
185 from collections import deque
186 def course_schedule(n, prerequisites):
187     graph = [[] for i in range(n)]
188     indegree = [0 for i in range(n)]
189     for pre in prerequisites:
190         graph[pre[1]].append(pre[0])
191         indegree[pre[0]] += 1
192     order = []
193     queue = deque([i for i in range(n) if indegree[i
    ] == 0])
194     while queue:
195         vertex = queue.popleft()
```

```
196         order.append(vertex)
197         for neighbor in graph[vertex]:
198             indegree[neighbor] -= 1
199             if indegree[neighbor] == 0:
200                 queue.append(neighbor)
201     return len(order) == n
202 
203 # 8- Kth permutation:
204 
205 import itertools
206 
207 def kth_permutation(n, k):
208     permutations = list(itertools.permutations(range
    (1, n+1)))
209     return ''.join(map(str, permutations[k-1]))
210 
211 
212 def kth_permutation(n, k):
213     permutation = []
214     unused = list(range(1, n+1))
215     fact = [1]*(n+1)
216     for i in range(1, n+1):
217         fact[i] = i*fact[i-1]
218     k -= 1
219     while n > 0:
220         part_length = fact[n]//n
221         i = k//part_length
222         permutation.append(unused[i])
223         unused.pop(i)
224         n -= 1
225         k %= part_length
226     return ''.join(map(str, permutation))
227 
228 
229 # 9- Minimum window substring:
230 
231 def contains_all(freq1, freq2):
232     for ch in freq2:
233         if freq1[ch] < freq2[ch]:
234             return False
235     return True
```

```
236
237
238 def min_window(s, t):
239     n, m = len(s), len(t)
240     if m > n or m == 0:
241         return ""
242     freqt = Counter(t)
243     shortest = " "*(n+1)
244     for length in range(1, n+1):
245         for i in range(n-length+1):
246             sub = s[i:i+length]
247             freqs = Counter(sub)
248             if contains_all(freqs, freqt) and length
     < len(shortest):
249                 shortest = sub
250     return shortest if len(shortest) <= n else ""
251 def min_window(s, t):
252     n, m = len(s), len(t)
253     if m > n or t == "":
254         return ""
255     freqt = Counter(t)
256     start, end = 0, n+1
257     for length in range(1, n+1):
258         freqs = Counter()
259         satisfied = 0
260         for ch in s[:length]:
261             freqs[ch] += 1
262             if ch in freqt and freqs[ch] == freqt[ch
    ]:
263                 satisfied += 1
264         if satisfied == len(freqt) and length < end-
    start:
265             start, end = 0, length
266         for i in range(1, n-length+1):
267             freqs[s[i+length-1]] += 1
268             if s[i+length-1] in freqt and freqs[s[i+
    length-1]] == freqt[s[i+length-1]]:
269                 satisfied += 1
270             if s[i-1] in freqt and freqs[s[i-1]] ==
    freqt[s[i-1]]:
271                 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]:
```

```
307             right += 1
308         max_area = max(max_area, heights[i]*(right-
    left+1))
309     return max_area
310 
311 
312 def rec(heights, low, high):
313     if low > high:
314         return 0
315     elif low == high:
316         return heights[low]
317     else:
318         minh = min(heights[low:high + 1])
319         pos_min = heights.index(minh, low, high + 1)
320         from_left = rec(heights, low, pos_min - 1)
321         from_right = rec(heights, pos_min + 1, high)
322         return max(from_left, from_right, minh * (
    high - low + 1))
323 
324 
325 def largest_rectangle(heights):
326     return rec(heights, 0, len(heights) - 1)
327 def largest_rectangle(heights):
328     heights = [-1]+heights+[-1]
329     from_left = [0]*len(heights)
330     stack = [0]
331     for i in range(1, len(heights)-1):
332         while heights[stack[-1]] >= heights[i]:
333             stack.pop()
334         from_left[i] = stack[-1]
335         stack.append(i)
336     from_right = [0]*len(heights)
337     stack = [len(heights)-1]
338     for i in range(1, len(heights)-1)[::-1]:
339         while heights[stack[-1]] >= heights[i]:
340             stack.pop()
341         from_right[i] = stack[-1]
342         stack.append(i)
343     max_area = 0
344     for i in range(1, len(heights)-1):
345         max_area = max(max_area, heights[i]*(
```

```
345 from_right[i]-from_left[i]-1))
346     return max_area
347 def largest_rectangle(heights):
348     heights = [-1]+heights+[-1]
349     max_area = 0
350     stack = [(0, -1)]
351     for i in range(1, len(heights)):
352         start = i
353         while stack[-1][1] > heights[i]:
354             top_index, top_height = stack.pop()
355             max_area = max(max_area, top_height*(i-
    top_index))
356             start = top_index
357         stack.append((start, heights[i]))
358     return max_area
359
360
361
362
363
364
```