1.Master’s theorem

Analyze $T_a(n) = 7\cdot T_a(n/2) + n^2$
$a = 7, b = 2, f(n) = n^2$
$\log_b(a) = \log_2(7) \approx 2.807$
Compare $f(n) = n^2$ with $n^{\log_2 7} \approx n^{2.807}$:
$n^2 = o(n^{2.807})$, so Case 1 applies
$\Rightarrow T_a(n) = \Theta(n^{\log_2 7})$
Analyze $T_b(n) = k\cdot T_b(n/4) + n^2$
$a = k, b = 4, f(n) = n^2$
$\log_4(k) = \log_2(k)/\log_2(4) = \log_2(k)/2$
Compare $f(n) = n^2$ with $n^{\log_4(k)}$:
We want $T_b(n) = O(T_a(n)) = O(n^{\log_2 7})$
So, to ensure $T_b(n) = O(n^{\log_2 7})$, we must ensure that the exponent $\log_4(k) \le \log_2 7$
Let’s express everything in base 2:
$\log_4(k) = (1/2)\cdot\log_2(k)$
So, set:
$(1/2)\cdot\log_2(k) \le \log_2(7)$
$\Rightarrow \log_2(k) \le 2\cdot\log_2(7)$
$\Rightarrow \log_2(k) \le \log_2(7^2) = \log_2(49)$
$\Rightarrow k \le 49$
Since k must be an integer, the maximum value of k is 49

2. You are given a list of n independent tasks, A = [a1, a2, . . . , an], where ai ∈ N represents the workload of ith task in units. A single core processor can process one task per time unit at a uniform rate k. When it works on task i for one time unit, it reduces the remaining workload by min (k, remaining workload of task i). A task with remaining workload zero is considered complete, and the machine may switch to any other incomplete task in the next time unit. The goal is to complete all tasks within at most h time units. **BINARY SEARCH DnQ**

```python
def min_rate(tasks, h):
    def can_finish(k):
        return sum((task + k - 1) // k for
                   task in tasks) <= h

    low, high = 1, max(tasks)
    while low < high:
        mid = (low + high) // 2
        if can_finish(mid):
            high = mid
        else:
            low = mid + 1
    return low
```

O(nlog(max ai))

3. Given a non-empty string s composed only of digits, count the number of valid ways to interpret it as a sequence of letters using the mapping:
1 → A, 2 → B, . . . , 26 → Z.
Rules:
• A single digit 1 – 9 can be map to a letter.
• Any two-digit number from 10 to 26 (inclusive) can map to letter.
• String with leading zeros (e.g., 06) are invalid. Write an O (n)-time algorithm to count the total number of valid decodings. If there are none, return 0. For example, 226 can be decoded into three ways, BZ, VF and BBF. **DYNAMIC PROGRAMMING**

```python
def num_decodings(s):
    if not s or s[0] == '0': return 0
    n = len(s)
    dp = [1] + [0] * n
    dp[1] = 1

    for i in range(2, n+1):
        one = int(s[i-1])
        two = int(s[i-2:i])
        if 1 <= one <= 9:
            dp[i] += dp[i-1]
        if 10 <= two <= 26:
            dp[i] += dp[i-2]
    return dp[n]
```

4. Given a list of non-negative integers. Write an O (nk log n)-time algorithm, where n is the number of elements in the list and k is the maximum length of the string representations of numbers, to arrange all these numbers in some order to create the numerically greatest possible concatenation when the numbers are written side by side. For example, [1,2,3] must be rearranged to 321 to maximize its numeric value. **GREEDY**

```python
from functools import cmp_to_key

def compare(x, y):
    return (int(y + x) > int(x + y))
    - (int(y + x) < int(x + y))

def largest_number(nums):
    nums = list(map(str, nums))
    nums.sort(key=cmp_to_key(compare))
    return '0' if nums[0] == '0' else ''.join(nums)
```

5. Consider there are n courses labeled from 0 to n − 1. You are given a list of prerequisites pairs: Prerequisites = [[c1, p1], [c2, p2], . . . , [cn, pn]], where [ci, pi] means you must complete course pi before taking course ci. Write an O (n + m)-time algorithm, where n is the number of courses and m is the number of prerequisite pairs, to find and return a valid order in which to complete all n courses so that every course is taken after all its prerequisites. If multiple valid orders exist, you may return any one. If no such order exists (for example, due to cyclic dependencies), return an empty list. **GRAPH TOPOLOGICAL SORT BFS**

```python
from collections import defaultdict, deque

def course_schedule(n, prerequisites):
    graph = defaultdict(list)
    indegree = [0] * n

    for c, p in prerequisites:
        graph[p].append(c)
        indegree[c] += 1

    queue = deque([i for i in range(n) if
                   indegree[i] == 0])
    order = []

    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return order if len(order) == n else []
```

6. Let P = {p1, p2, . . . , pn} ⊂ Z2 be a finite set of n distinct points in the two-dimensional integer lattice, where each point is represented as pi = (xi, yi). For any pair of points pi and pj, define the cost of connecting them as their Manhattan distance:
cost(pi, pj) = |xi − xj| + |yi − yj|.
You are permitted to construct a set of connections (i.e., undirected edges) between selected pairs of points in P, such that the following conditions are satisfied:
• The resulting structure is connected; that is, for any pair of points pi and pj, there exists a sequence of connected edges linking them.
• The total cost, defined as the sum of the costs of all selected connections, is minimized.
Write an O(n^2 log n)-time algorithm to determine the minimum possible total cost required to connect all points in P under these conditions.
GRAPH PRIM’S ALGORITHM WITH MANHATTAN DISTANCE

```python
import heapq

def min_cost_connect_points(points):
    n = len(points)
    visited = set()
    min_heap = [(0, 0)]  # cost, index
    res = 0

    while len(visited) < n:
        cost, u = heapq.heappop(min_heap)
        if u in visited:
            continue
        visited.add(u)
        res += cost
        for v in range(n):
            if v not in visited:
                dist = abs(points[u][0] - points[v][0]) +
                abs(points[u][1] - points[v][1])
                heapq.heappush(min_heap, (dist, v))
    return res
```