```python
from typing import List, Tuple
import heapq
from collections import deque

# 1) Insertion Sort
def insertion_sort(arr: List[int]) -> List[int]:
    a = arr[:]
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
    return a

# 2) Selection Sort
def selection_sort(arr: List[int]) -> List[int]:
    a = arr[:]
    for i in range(len(a)):
        min_index = i
        for j in range(i + 1, len(a)):
            if a[j] < a[min_index]:
                min_index = j
        a[i], a[min_index] = a[min_index], a[i]
    return a

# 3) Factorial
def factorial_iterative(n: int) -> int:
    if n < 0: raise ValueError("Negative numbers not allowed")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def factorial_recursive(n: int) -> int:
    if n < 0: raise ValueError("Negative numbers not allowed")
    if n <= 1: return 1
    return n * factorial_recursive(n - 1)

# 4) Fibonacci
def fibonacci_recursive(n: int) -> int:
    if n < 0: raise ValueError("Invalid input")
    if n <= 1: return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

def fibonacci_iterative(n: int) -> int:
    if n < 0: raise ValueError("Invalid input")
    if n <= 1: return n
    prev2, prev1 = 0, 1
    for _ in range(2, n + 1):
        prev2, prev1 = prev1, prev1 + prev2
    return prev1

# 5) Linear Search
def linear_search(arr: List[int], target: int) -> int:
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# 6) QuickSort Variants
class QuickSort:
    @staticmethod
    def quicksort_right(a: List[int], l: int, r: int):
        if l < r:
            p = QuickSort.partition_right(a, l, r)
            QuickSort.quicksort_right(a, l, p - 1)
            QuickSort.quicksort_right(a, p + 1, r)

    @staticmethod
    def partition_right(a: List[int], l: int, r: int) -> int:
        pivot = a[r]
        i = l
        for j in range(l, r):
            if a[j] <= pivot:
                a[i], a[j] = a[j], a[i]
                i += 1
        a[i], a[r] = a[r], a[i]
        return i

    @staticmethod
    def quicksort_left(a: List[int], l: int, r: int):
        if l < r:
            p = QuickSort.partition_left(a, l, r)
            QuickSort.quicksort_left(a, l, p - 1)
            QuickSort.quicksort_left(a, p + 1, r)

    @staticmethod
    def partition_left(a: List[int], l: int, r: int) -> int:
        pivot = a[l]
        i, j = l + 1, r
        while True:
            while i <= j and a[i] < pivot: i += 1
            while i <= j and a[j] > pivot: j -= 1
            if i >= j: break
            a[i], a[j] = a[j], a[i]
            i += 1; j -= 1
        a[l], a[j] = a[j], a[l]
        return j

    @staticmethod
    def quicksort_median(a: List[int], l: int, r: int):
        if l < r:
            p = QuickSort.partition_median(a, l, r)
            QuickSort.quicksort_median(a, l, p - 1)
            QuickSort.quicksort_median(a, p + 1, r)

    @staticmethod
    def partition_median(a: List[int], l: int, r: int) -> int:
        mid = l + (r - l) // 2
        pivot_index = QuickSort.median_of_three(a, l, mid, r)
        a[pivot_index], a[r] = a[r], a[pivot_index]
        pivot = a[r]
        i = l
        for j in range(l, r):
            if a[j] <= pivot:
                a[i], a[j] = a[j], a[i]
                i += 1
        a[i], a[r] = a[r], a[i]
        return i

    @staticmethod
```

```
    def median_of_three(a: List[int], i: int, j: int, k: int) -> int:
        if (a[i] - a[j]) * (a[k] - a[i]) >= 0: return i
        if (a[j] - a[i]) * (a[k] - a[j]) >= 0: return j
        return k

    @staticmethod
    def quicksort_left_duplicates(a: List[int], l: int, r: int):
        if l < r:
            i, k = QuickSort.partition3(a, l, r)
            QuickSort.quicksort_left_duplicates(a, l, i - 1)
            QuickSort.quicksort_left_duplicates(a, k + 1, r)

    @staticmethod
    def partition3(a: List[int], l: int, r: int) -> Tuple[int,int]:
        pivot = a[l]
        i = j = l; k = r
        while j <= k:
            if a[j] < pivot:
                a[i], a[j] = a[j], a[i]
                i += 1; j += 1
            elif a[j] > pivot:
                a[j], a[k] = a[k], a[j]
                k -= 1
            else:
                j += 1
        return i, k

# 7) HeapSort
class HeapSort:
    @staticmethod
    def heap_sort(arr: List[int]):
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            HeapSort.max_heapify(arr, i, n)
        for i in range(n - 1, 0, -1):
            arr[0], arr[i] = arr[i], arr[0]
            HeapSort.max_heapify(arr, 0, i)

    @staticmethod
    def max_heapify(arr: List[int], i: int, n: int):
        largest = i
        left, right = 2*i + 1, 2*i + 2
        if left < n and arr[left] > arr[largest]: largest = left
        if right < n and arr[right] > arr[largest]: largest = right
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            HeapSort.max_heapify(arr, largest, n)

class MinHeapSort:
    @staticmethod
    def min_heap_sort(arr: List[int]):
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            MinHeapSort.min_heapify(arr, i, n)
        for i in range(n - 1, 0, -1):
            arr[0], arr[i] = arr[i], arr[0]
            MinHeapSort.min_heapify(arr, 0, i)

    @staticmethod
    def min_heapify(arr: List[int], i: int, n: int):
        smallest = i
        left, right = 2*i + 1, 2*i + 2
        if left < n and arr[left] < arr[smallest]: smallest = left
        if right < n and arr[right] < arr[smallest]: smallest = right
        if smallest != i:
            arr[i], arr[smallest] = arr[smallest], arr[i]
            MinHeapSort.min_heapify(arr, smallest, n)

# 8) BFS
def bfs(adj: List[List[int]], start: int) -> List[int]:
    visited = [False] * len(adj)
    q = deque([start])
    visited[start] = True
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append(v)
    return order

# 9) Kruskal MST
class Kruskal:
    class Edge:
        def __init__(self, u, v, w): self.u, self.v, self.w = u, v, w
        def __lt__(self, other): return self.w < other.w

    class DSU:
        def __init__(self, n):
            self.parent = list(range(n))
            self.rank = [0]*n
        def find(self, x):
            if self.parent[x] != x: self.parent[x] = self.find(self.parent[x])
            return self.parent[x]
        def union(self, a, b):
            ra, rb = self.find(a), self.find(b)
            if ra == rb: return False
            if self.rank[ra] < self.rank[rb]: self.parent[ra] = rb
            elif self.rank[ra] > self.rank[rb]: self.parent[rb] = ra
            else: self.parent[rb] = ra; self.rank[ra] += 1
            return True

    @staticmethod
    def kruskal_mst(n: int, edges: List['Kruskal.Edge']) -> List['Kruskal.Edge']:
        edges.sort()
        dsu = Kruskal.DSU(n)
        mst = []
        for e in edges:
            if dsu.union(e.u, e.v): mst.append(e)
        return mst

# 10) Prim MST
def prim_mst(n: int, edges: List[Tuple[int,int,int]]) -> int:
    adj = [[] for _ in range(n)]
    for u,v,w in edges:
        adj[u].append((v,w)); adj[v].append((u,w))
    visited = [False]*n
    pq = [(w,v) for v,w in adj[0]]
    heapq.heapify(pq)
    visited[0] = True
    total = 0
```

```
    while pq:
        w,v = heapq.heappop(pq)
        if visited[v]: continue
        visited[v] = True
        total += w
        for u,weight in adj[v]:
            if not visited[u]: heapq.heappush(pq, (weight,u))
    return total

# 11) Floyd-Warshall
def floyd_warshall(dist: List[List[float]]) -> List[List[float]]:
    n = len(dist)
    d = [row[:] for row in dist]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]
    return d

# 12) 0/1 Knapsack
def knapsack_01(values: List[int], weights: List[int], capacity: int) -> int:
    n = len(values)
    dp = [[0]*(capacity+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(capacity+1):
            dp[i][w] = dp[i-1][w]
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i][w], values[i-1] + dp[i-1][w - weights[i-1]])
    return dp[n][capacity]
```