**1.)**

Claim:

For any integer $k \geq 1$,

$\sum_{i=1}^{n} i^k \cdot \log i = O(n^{k+1} \cdot \log n)$

Proof (by integral approximation):

Let $S(n) = \sum_{i=1}^{n} i^k \cdot \log i$

Use integral approximation:

$S(n) \leq \int_1^n x^k \log x \, dx + n^k \log n$

Now compute the integral:

Let $u = \log x$, $dv = x^k \, dx$

Then:

$du = 1/x \, dx$, $v = x^{k+1}/(k+1)$

So:

$\int x^k \log x \, dx = (x^{k+1}/(k+1)) \cdot \log x - \int x^k \, dx / (k+1)$

$= (x^{k+1}/(k+1)) \cdot \log x - (1/(k+1)^2) \cdot x^{k+1}$

Evaluating from 1 to n:

$\int_1^n x^k \log x \, dx = (n^{k+1}/(k+1)) \cdot \log n - (n^{k+1}/(k+1)^2)$ + constants

$\Rightarrow \Theta(n^{k+1} \log n)$

Thus:

$S(n) = \sum_{i=1}^{n} i^k \log i = O(n^{k+1} \log n)$ ☐

**2a(n)**: s := 0

```
for i := 0 to n do
  for j := 1 to i do
    for k := j to i + j do
      s := s + 1
```

Analysis:

Innermost loop runs from k = j to k = i + j, hence it iterates i + 1 times. For each fixed i, the middle loop runs i times.

Thus, Total iterations = $\sum$(i=0 to n) i * (i + 1)

= $\sum$(i=0 to n) ($i^2$ + i) = $O(n^3)$.

**2b(n)**:

```
i := 0
j := n
s := 0
while i < j do:
  i := i + 2
  j := j - 3
  s := s + 1
```

Analysis:

After t iterations, i = 2t, j = n - 3t. Loop stops when i ≥ j.

Solving for t, we get t ≥ n/5.

Thus, total iterations = O(n).

Time Complexity: T(n) = O(n)

Procedure Problem**2c(n)**:

```
if n ≤ 0 then return
for i := 1 to n do:
  for j := 1 to n do:
    s := s + 1
for i := 1 to 4 do:
  Problem2c(n / 4)
```

$$S = \sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$$

$$T(n) = n^2 + \frac{4n^2}{16} + \frac{16n^2}{256} + 64T\left(\frac{n}{64}\right)$$

$$T(n) = n^2 + 4T\left(\frac{n}{4}\right)$$

$\Theta(n^2)$ or $O(n^2)$

**3.** Given n arrays, each array contain n positive integers. Write an O n2 log n algorithm to find the smallest n sums out of nn possible sums that can be obtained by picking one positive integer from each of n arrays. For example, given three arrays as follows: [5, 1, 8] , [5, 2, 9] , and [6, 7, 10]. The smallest n sums of the given array is [9, 10, 12].

```
SmallestNSums(Arrays):
    Let n = length(Arrays)
    For each array A in Arrays:
        Sort A in ascending order
    Set CurrentSums = Arrays[0]
    For i = 1 to n - 1:
        CurrentSums = MergeTwoArrays(CurrentSums, Arrays[i], n)
    Return CurrentSums

MergeTwoArrays(A, B, n):
    Initialize MinHeap H
    Initialize VisitedPairs as an empty set
    Initialize Result as an empty list
    Push (A[0] + B[0], 0, 0) into H
    Add (0, 0) to VisitedPairs
    While length(Result) < n:
        Pop the smallest element (sum_val, i, j) from H
        Append sum_val to Result
        If i + 1 < n and (i + 1, j) not in VisitedPairs:
            Push (A[i + 1] + B[j], i + 1, j) into H
            Add (i + 1, j) to VisitedPairs
        If j + 1 < n and (i, j + 1) not in VisitedPairs:
            Push (A[i] + B[j + 1], i, j + 1) into H
            Add (i, j + 1) to VisitedPairs
    Return Result
```

**4.** Write an algorithm to sort n integers in the range 0 to n3 – 1 in O (n) time.

```
def counting_sort(arr, n):
    # Step 1: Initialize a count array of size n^3
    # (since values are between 0 and n^3-1)
    count = [0] * (n**3)

    # Step 2: Count the frequency of each integer in arr
    for i in range(n):
        count[arr[i]] += 1

    # Step 3: Reconstruct the sorted array
    index = 0
    for value in range(n**3):
        while count[value] > 0:
            arr[index] = value
            index += 1
            count[value] -= 1

    return arr
```

**5.** ssuming that there are no duplicate keys in a binary search tree, develop an algorithm to determine the pre-order traversal of the tree based on their in-order and post-order traversal. For example,

In-order traversal: 3 1 4 0 2 5

Post-order traversal: 3 4 1 5 2 0

Output: 0 1 3 4 2 5

```
def PreOrderFromInPost(inorder, postorder):
    if not inorder:
        return []

    # Root is the last element of postorder
    root = postorder[-1]
    root_index = inorder.index(root)

    # Left and right inorder and postorder arrays
    left_inorder = inorder[:root_index]
    right_inorder = inorder[root_index + 1:]
    left_postorder = postorder[:len(left_inorder)]
    right_postorder = postorder[len(left_inorder):-1]

    # Recursively find the pre-order traversal
    left_preorder = PreOrderFromInPost(left_inorder, left_postorder)
    right_preorder = PreOrderFromInPost(right_inorder, right_postorder)

    # Return root followed by left and right pre-orders
    return [root] + left_preorder + right_preorder

# Example usage
inorder = [3, 1, 4, 0, 2, 5]
postorder = [3, 4, 1, 5, 2, 0]
preorder = PreOrderFromInPost(inorder, postorder)
print(preorder)  # Output: [0, 1, 3, 4, 2, 5]
```

**6.** Write the algorithm Tree-Delete(T, z) to delete a node z from a Binary Search Tree. When z has two children, use its Predecessor (maximum in the left subtree) to replace it instead of the Successor.

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None

def tree_delete(T, z):
    if z.left is None and z.right is None:
        if z.parent is None:
            T.root = None
        else:
            if z.parent.left == z:
                z.parent.left = None
            else:
                z.parent.right = None
    elif z.left is None or z.right is None:
        if z.left is not None:
            child = z.left
        else:
            child = z.right

        if z.parent is None:
            T.root = child
        else:
            if z.parent.left == z:
                z.parent.left = child
            else:
                z.parent.right = child
        child.parent = z.parent
    else:
        y = tree_predecessor(z)

        if y.parent != z:
            if y.left is not None:
                y.parent.right = y.left
                y.left.parent = y.parent
            else:
                y.parent.right = None

        if z.parent is None:
            T.root = y
        else:
            if z.parent.left == z:
                z.parent.left = y
            else:
                z.parent.right = y

        y.left = z.left
        if y.left is not None:
            y.left.parent = y

        y.right = z.right
        if y.right is not None:
            y.right.parent = y

        y.parent = z.parent
```

```python
def tree_predecessor(z):
    if z.left is not None:
        return tree_maximum(z.left)
    else:
        return None

def tree_maximum(x):
    while x.right is not None:
        x = x.right
    return x
```

(7) The $d$-ary heap, also known as the $d$-heap, is a type of priority queue data structure. It extends the concept of the binary heap by allowing nodes to have $d$ children instead of just 2.

(a) Describe how to represent a $d$-ary heap in an array.

(b) Give an efficient implementation of EXTRACTMIN in a $d$-ary min-heap. Analyze its running time in terms of $d$ and $n$.

(c) Give an efficient implementation of DECREASEKEY in a $d$-ary min-heap. Analyze its running time in terms of $d$ and $n$.

(d) Give an efficient implementation of INSERT in a $d$-ary min-heap. Analyze its running time in terms of $d$ and $n$.

**a).** A d-ary heap is stored in an array, just like a binary heap, but each node has up to d children.

For a node at index i (0-based indexing):

- Its parent is at index $\lfloor (i - 1) / d \rfloor$
- Its children are at indices $d \cdot i + 1$ to $d \cdot i + d$

This lets us efficiently store and access the heap without using pointers.

```python
    def decrease_key(self, i, new_value):
        if new_value > self.heap[i]:
            return
        self.heap[i] = new_value
        while i > 0 and self.heap[(i - 1) // self.d] > self.heap[i]:
            parent_index = (i - 1) // self.d
            self.heap[i], self.heap[parent_index] = self.heap[parent_index], self.heap[i]
            i = parent_index

    def insert(self, value):
        self.heap.append(value)
        i = len(self.heap) - 1
        self.decrease_key(i, value)

# Example Usage:
heap = DaryHeap(d=3)  # 3-ary heap
heap.insert(10)
heap.insert(20)
heap.insert(5)
print(heap.extract_min())  # Output: 5
heap.decrease_key(1, 3)
print(heap.extract_min())  # Output: 3
```

```python
class DaryHeap:
    def __init__(self, d):
        self.d = d
        self.heap = []

    def extract_min(self):
        if len(self.heap) == 0:
            return None
        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        min_element = self.heap.pop()
        self._heapify_down(0)
        return min_element

    def _heapify_down(self, i):
        size = len(self.heap)
        min_index = i
        for j in range(1, self.d + 1):
            child_index = self.d * i + j
            if child_index < size and self.heap[child_index] < self.heap[min_index]:
                min_index = child_index
        if min_index != i:
            self.heap[i], self.heap[min_index] = self.heap[min_index], self.heap[i]
            self._heapify_down(min_index)
```