DSA ALGORITHM

Stack Operations Algorithm

1. Initialize Stack

```
Algorithm init_stack(capacity):
    create stack array of given capacity
    set top = -1  // indicates empty stack
```

2. Push (Insert)

```
Algorithm push(stack, item):
    if stack is full (top == capacity - 1):
        return "Stack Overflow"
    else:
        increment top by 1
        stack[top] = item
```

3. Pop (Remove)

```
Algorithm pop(stack):
    if stack is empty (top == -1):
        return "Stack Underflow"
    else:
        item = stack[top]
        decrement top by 1
        return item
```

4. Peek (Top Element)

```
```

```
Algorithm peek(stack):
    if stack is empty:
        return "Stack is Empty"
    else:
        return stack[top]
```

5. Check if Empty

```
Algorithm is_empty(stack):
    return (top == -1)
```

6. Check if Full

```
Algorithm is_full(stack, capacity):
    return (top == capacity - 1)
```

7. Display (Print Stack)

```
Algorithm display(stack):
    if stack is empty:
        print "Stack is empty"
    else:
        print "Stack (Top to Bottom):"
        for i from top down to 0:
            print stack[i]
```

Queue Operations Algorithm

(FIFO - First In First Out)

1. Initialize Queue

```
Algorithm init_queue(capacity):
    create queue array of given capacity
    set front = -1    // indicates empty queue
    set rear = -1
```

2. Enqueue (Insert at Rear)

```
Algorithm enqueue(queue, item):
    if queue is full (rear == capacity - 1):
        return "Queue Overflow"
    else:
        increment rear by 1
        queue[rear] = item
```

3. Dequeue (Remove from Front)

```
Algorithm dequeue(queue):
    if queue is empty (front == rear):
        return "Queue Underflow"
    else:
        increment front by 1
        item = queue[front]
        if front == rear:  // reset when last element is removed
            set front = -1
            set rear = -1
        return item
```

5. Check if Empty

```
Algorithm is_empty(queue):
    return (front == rear)
```

6. Check if Full

```
Algorithm is_full(queue, capacity):
    return (rear == capacity - 1)
```

7. Display (Print Queue)

```
Algorithm display(queue):
    if queue is empty:
        print "Queue is empty"
    else:
        print "Queue (Front to Rear):"
        for i from front + 1 to rear:
            print queue[i]
```

8. Size (Number of Elements)

```
Algorithm size(queue):
    if queue is empty:
        return 0
    else:
        return (rear - front)
```

CIRCULAR QUEUE

Algorithm ENQUEUE(cqueue, item):

```
    if (rear + 1) mod MAX == front:
        print "QUEUE OVERFLOW"
    else:
        cqueue[rear] ← item
        rear ← (rear + 1) mod MAX
        print "One item added"
```

Algorithm DEQUEUE(cqueue):

```
    if front == rear:
        print "Empty Queue"
    else:
        item ← cqueue[front]
        front ← (front + 1) mod MAX
        print "Deleted item =", item
```

Algorithm DISPLAY(cqueue):

```
   if front == rear:
        print "Empty Queue"
    else:
        i ← front
        while i ≠ rear:
            print cqueue[i]
            i ← (i + 1) mod MAX
```

Algorithm isEmpty(cqueue):

```
Algorithm isEmpty(cqueue):
    if front == rear:
        return TRUE
    else:
        return FALSE
```

Algorithm isFull(cqueue):

```
Algorithm isFull(cqueue):
    if (rear + 1) mod MAX == front:
        return TRUE
    else:
        return FALSE
```

Algorithm size(cqueue):

```
Algorithm size(cqueue):
    return (rear - front + MAX) mod MAX
```

DEQUE (DOUBLE-ENDED QUEUE)

Algorithm INSERTION_REAR_END(deque, item):

```
Algorithm INSERTION_REAR_END(deque, item):
    if REAR == MAX - 1:
        PRINT "Insertion Not Possible (Queue Overflow)"
    else:
        REAR ← REAR + 1
        deque[REAR] ← item
        PRINT "One item added at rear"
```

Algorithm INSERTION_FRONT_END(deque, item):

```
Algorithm INSERTION_FRONT_END(deque, item):
    if FRONT == -1:
        PRINT "Insertion Not Possible (Queue Underflow)"
    else:
        deque[FRONT] ← item
        FRONT ← FRONT - 1
        PRINT "One item added at front"
```

Algorithm DELETION_FRONT_END(deque):

```
Algorithm DELETION_FRONT_END(deque):
    if REAR == FRONT:
        PRINT "Empty Deque (Underflow)"
    else:
        FRONT ← FRONT + 1
        item ← deque[FRONT]
        PRINT "Deleted item =", item
```

Algorithm DELETION_REAR_END(deque):

```
Algorithm DELETION_REAR_END(deque):
    if REAR == FRONT:
        PRINT "Empty Deque (Underflow)"
    else:
        item ← deque[REAR]
        REAR ← REAR - 1
        PRINT "Deleted item =", item
```

Algorithm DISPLAY(deque):

```
Algorithm DISPLAY(deque):
    if REAR == FRONT:
```

```
        PRINT "Empty Deque"
    else:
        PRINT "The deque is given below:"
        i ← FRONT + 1
        WHILE i ≤ REAR:
            PRINT deque[i]
            i ← i + 1
```

Algorithm ISEMPTY(deque):

```
Algorithm ISEMPTY(deque):
    if FRONT == REAR:
        RETURN TRUE
    else:
        RETURN FALSE
```

Algorithm ISFULL(deque):

```
Algorithm ISFULL(deque):
    if REAR == MAX - 1:
        RETURN TRUE
    else:
        RETURN FALSE
```

Algorithm SIZE(deque):

```
Algorithm SIZE(deque):
    RETURN REAR - FRONT
```

Linked List (Singly)

Algorithm INSERT_BEGIN(head, item):

```
Algorithm INSERT_BEGIN(head, item):
    new_node ← CREATE_NODE(item)
    new_node.next ← head
    head ← new_node
    RETURN head
```

Algorithm INSERT_END(head, item):

```
Algorithm INSERT_END(head, item):
    new_node ← CREATE_NODE(item)
    new_node.next ← NULL

    if head == NULL:
        head ← new_node
    else:
        temp ← head
        WHILE temp.next ≠ NULL:
            temp ← temp.next
```

```
        temp.next ← new_node

    RETURN head
```

Algorithm INSERT_AT_POSITION OR UNIVERSAL INSERTION (head, item, position):

```
Algorithm INSERT_AT_POSITION(head, item, position):
    new_node ← CREATE_NODE(item)

    if position == 1:
        new_node.next ← head
        head ← new_node
    else:
        temp ← head
        count ← 1
        WHILE count < position − 1 AND temp ≠ NULL:
            temp ← temp.next
            count ← count + 1

        if temp == NULL:
            PRINT "Position out of bounds"
        else:
            new_node.next ← temp.next
            temp.next ← new_node

    RETURN head
```

Algorithm DELETE_NODE OR UNIVERSAL INSERTION (head, item):

```
    temp ← head

    if temp == NULL:
        PRINT "Empty List"
        RETURN head

    else if temp.data == item:
        PRINT "Deleted item:", temp.data
        head ← temp.next
        RETURN head

    prev ← temp
    temp ← temp.next

    WHILE temp ≠ NULL:
        if temp.data == item:
            prev.next ← temp.next
            PRINT "Deleted item:", temp.data
            RETURN head
        prev ← temp
        temp ← temp.next

    PRINT "Item Not found"
```

```
    RETURN head
```

Algorithm: DISPLAY(head)

```
    temp ← head

    if temp == NULL:
        PRINT "Empty List"
        RETURN

    WHILE temp ≠ NULL:
        PRINT temp.data
        temp ← temp.next
```

Algorithm: COUNT(head)

```
    temp ← head
    c ← 0

    if temp == NULL:
        PRINT "Empty List"
        RETURN c

   WHILE temp ≠ NULL:
        c ← c + 1
        temp ← temp.next

    RETURN c
```

Algorithm: SEARCH(head, item)

```
    temp ← head
    pos ← 0

    if temp == NULL:
        PRINT "Empty List"
        RETURN 0

    WHILE temp ≠ NULL:
        pos ← pos + 1
        if temp.data == item:
            RETURN pos
        temp ← temp.next

    RETURN -1
```

Bubble Sort

```
bubbleSort(array)
  for i <- 1 to sizeOfArray - 1
    for j <- 1 to sizeOfArray - 1 - i
      if leftElement > rightElement
        swap leftElement and rightElement
end bubbleSort
```

Selection Sort

```
1. Repeat for i from 0 to size − 1:
    a. Set minIndex = i
    b. For j from i + 1 to size − 1:
        i. If arr[j] < arr[minIndex]:
            Set minIndex = j
    c. Swap arr[i] and arr[minIndex]
```

Insertion Sort

```
1. Repeat for i from 1 to n − 1:
    a. key = arr[i]
    b. j = i − 1
    c. While j ≥ 0 and arr[j] > key:
        i. arr[j + 1] = arr[j]
        ii. j = j − 1
    d. arr[j + 1] = key
```

```
Algorithm MERGE_SORT(array):
    if LENGTH(array) > 1 then
        mid ← LENGTH(array) // 2
        leftHalf ← array[0 to mid − 1]
        rightHalf ← array[mid to LENGTH(array) − 1]

        MERGE_SORT(leftHalf)
        MERGE_SORT(rightHalf)

        i ← 0, j ← 0, k ← 0

        // Merge leftHalf and rightHalf into array
        while i < LENGTH(leftHalf) AND j < LENGTH(rightHalf) do
            if leftHalf[i] < rightHalf[j] then
                array[k] ← leftHalf[i]
                i ← i + 1
            else
                array[k] ← rightHalf[j]
                j ← j + 1
            k ← k + 1

        // Copy any remaining elements of leftHalf
        while i < LENGTH(leftHalf) do
            array[k] ← leftHalf[i]
            i ← i + 1
            k ← k + 1

        // Copy any remaining elements of rightHalf
        while j < LENGTH(rightHalf) do
            array[k] ← rightHalf[j]
            j ← j + 1
```

```
            k ← k + 1
```

QUICK SORT ALGO

```
QuickSort(arr, low, high)
    if low < high then
        pivot_index = Partition(arr, low, high)
        QuickSort(arr, low, pivot_index - 1)
        QuickSort(arr, pivot_index + 1, high)

Partition(arr, low, high)
    pivot = arr[high]
    i = low - 1
    for j from low to high - 1 do
        if arr[j] <= pivot then
            i = i + 1
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

B-Tree Creation (Text-based Algorithm)

1. Start with an empty tree.
2. Create a root node and mark it as a leaf.
3. Initialize it with zero keys.
4. Set the minimum degree t (defines max and min number of keys per node).
5. The tree is now ready for insertions.

B-Tree Insertion (Text-based Algorithm)

Check if root is full (i.e., has 2t - 1 keys):

If yes, create a new node S and make it the new root.

1. Make the old root its child.
2. Split the old root into two nodes.
3. Move the middle key of the old root up to the new root.
4. Now insert the key into the correct subtree.

If the root is not full, insert the key directly:

1. Traverse down to the correct child node.
2. If the child is full, split it before proceeding.
3. Continue this process recursively until the key is inserted into a non-full leaf node.

Always maintain the order of keys in nodes.

ALGORITHM BuildMaxHeap(A, size)

```
FOR i ← (size ÷ 2) − 1 DOWNTO 0 DO
    MaxHeapify(A, i, size)
```

ALGORITHM MaxHeapify(A, i, size)

```
largest ← i
left ← 2 × i + 1
right ← 2 × i + 2

IF left < size AND A[left] > A[largest] THEN
    largest ← left

IF right < size AND A[right] > A[largest] THEN
    largest ← right

IF largest ≠ i THEN
    SWAP A[i] and A[largest]
    MaxHeapify(A, largest, size)
```

ALGORITHM BuildMinHeap(A, size)

```
FOR i ← (size ÷ 2) − 1 DOWNTO 0 DO
    MinHeapify(A, i, size)
```

ALGORITHM MinHeapify(A, i, size)

```
smallest ← i
left ← 2 × i + 1
right ← 2 × i + 2
```

```
  IF left < size AND A[left] < A[smallest] THEN
      smallest ← left

  IF right < size AND A[right] < A[smallest] THEN
      smallest ← right

  IF smallest ≠ i THEN
      SWAP A[i] and A[smallest]
      MinHeapify(A, smallest, size)
```

Algorithm for insertion in Max Heap

```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

Algorithm for deletion in Max Heap

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

```
BREADTH-FIRST SEARCH (BFS) Algorithm

INPUT: Graph G(V, E), starting node start
OUTPUT: Nodes visited in BFS order


---
```

ALGORITHM BFS(G, start):

1. INITIALIZE a queue Q

2. MARK all nodes as unvisited

3. MARK start as visited

4. ENQUEUE start into Q

5. WHILE Q is not empty:
    a. current ← DEQUEUE(Q)
    b. PRINT current
    c. FOR EACH neighbor n of current DO
        i. IF n is not visited THEN
            A. MARK n as visited
            B. ENQUEUE n into Q

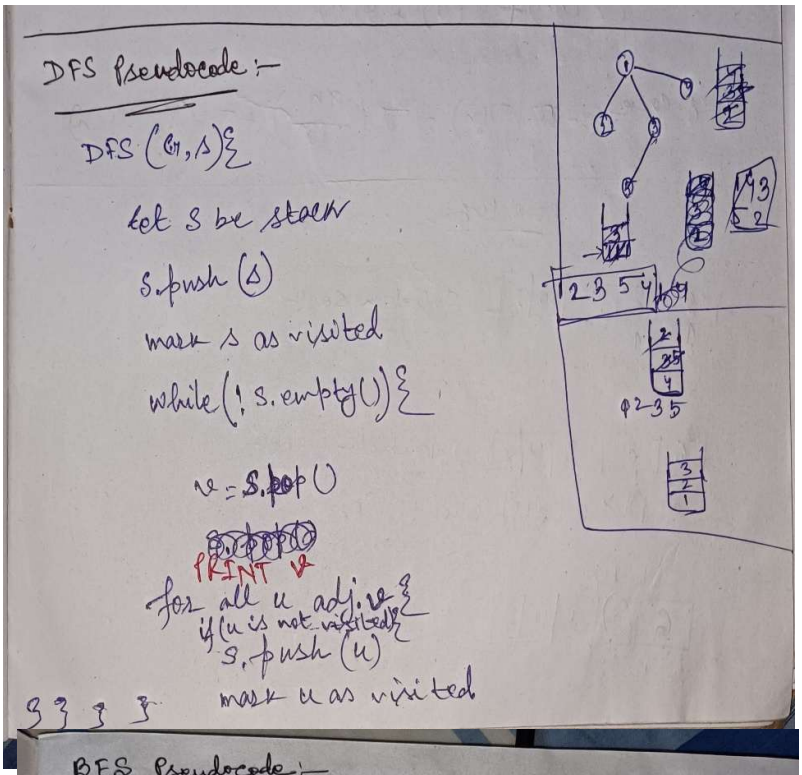
6. END WHILE

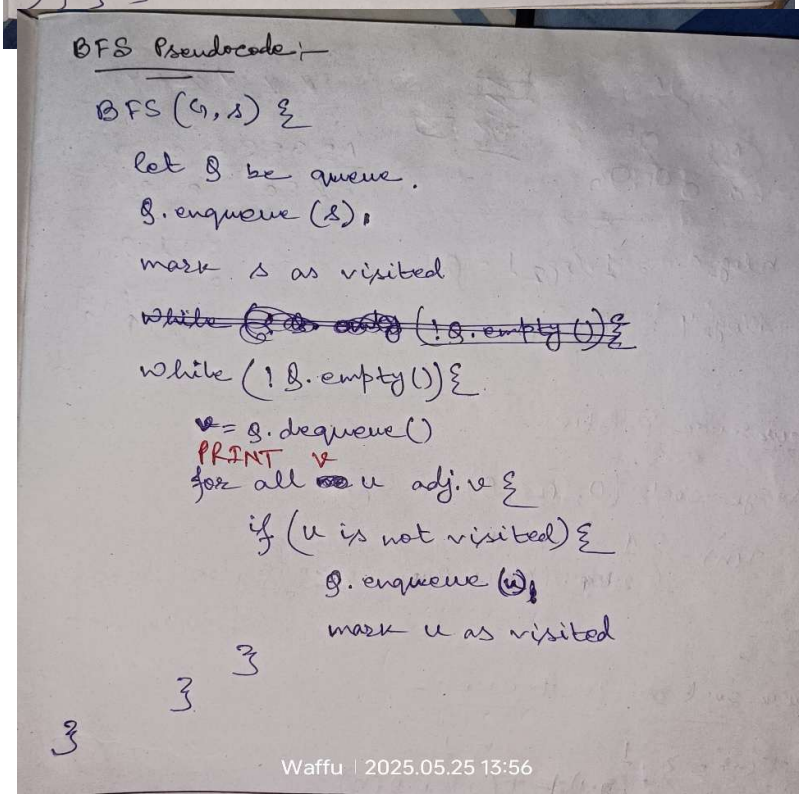
ALGORITHM DFS(G, start):

1. INITIALIZE an empty stack S

2. MARK all nodes as unvisited

3. PUSH start onto S





```
4. MARK start as visited


5. WHILE S is not empty:
        a. current ← POP(S)
        b. PRINT current
        c. FOR EACH neighbor n of current (in reverse order if needed):
                A. IF n is not visited THEN
                        1. PUSH n onto S
                        2. MARK n as visited

6. END WHILE
```

PRIM'S ALGORITHM FOR MCST/MST:

1. Initialize the minimum spanning tree with a single vertex selected at random from the graph.

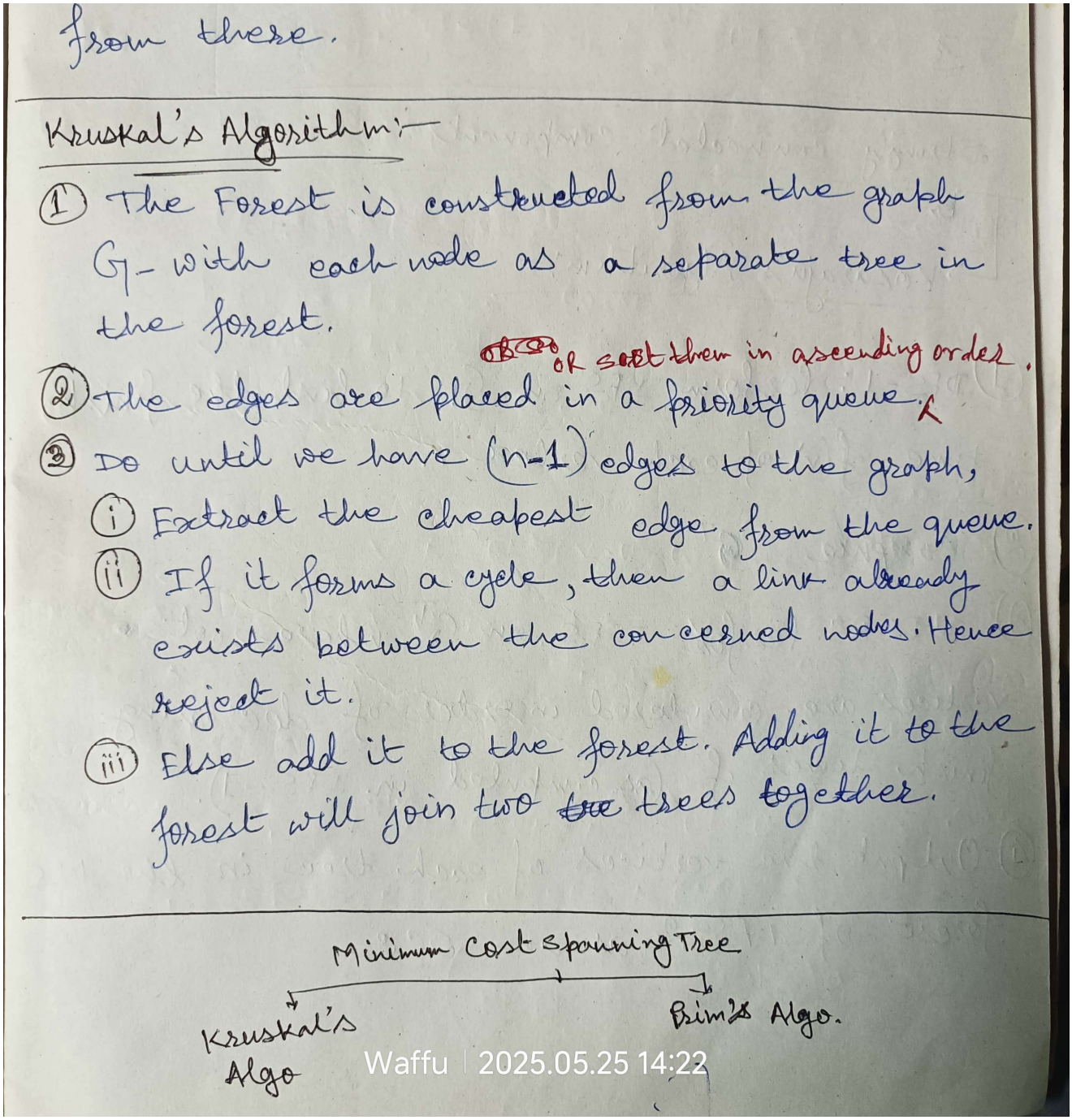
2. Find all edges that connect the tree to vertices that are not yet in the tree.

3. Select the edge with the smallest weight from the set of edges found in step 2 and add it to the minimum spanning tree.

4. Add the new vertex that is connected to the selected edge to the minimum spanning tree.

5. Repeat steps 2–4 until all vertices are part of the minimum spanning tree.

KRUSKAL'S ALGO FOR MCST/MST



from these.

Kruskal's Algorithm:-

1. The Forest is constructed from the graph G - with each node as a separate tree in the forest.
2. The edges are placed in a priority queue. OR sort them in ascending order.
3. Do until we have (n-1) edges to the graph,
   (i) Extract the cheapest edge from the queue.
   (ii) If it forms a cycle, then a link already exists between the concerned nodes. Hence reject it.
   (iii) Else add it to the forest. Adding it to the forest will join two ~~tre~~ trees ~~to~~gether.

Minimum Cost Spanning Tree

Kruskal's Algo

Prim's Algo.

Waffu | 2025.05.25 14:22

HEAP_SORT(arr):

```
    n ← length of arr

    // Step 1: Build Max Heap
    FOR i ← (n / 2) - 1 DOWNTO 0 DO
        HEAPIFY(arr, n, i)
```

```
    // Step 2: Extract elements from heap
    FOR i ← n − 1 DOWNTO 1 DO
        SWAP arr[0] WITH arr[i]         // Move max to end
        HEAPIFY(arr, i, 0)              // Heapify reduced heap
```

HEAPIFY(arr, n, i):

```
    largest ← i                         // Initialize largest as root
    left ← 2 * i + 1                    // Left child index
    right ← 2 * i + 2                   // Right child index

    IF left < n AND arr[left] > arr[largest] THEN
        largest ← left

    IF right < n AND arr[right] > arr[largest] THEN
        largest ← right

    IF largest ≠ i THEN
        SWAP arr[i] WITH arr[largest]  // Swap root with largest
        HEAPIFY(arr, n, largest)       // Recursively heapify subtree
```