1) Before we attempt to implement the functions that were asked (**popmin(A, n)**, **insert(A, n, value)**, and **sort(A, size)**), we need to implement some helper functions.

Helper functions:

```
/**
* This is a helper function. It percolates up the Matrix.
*/
function percolateUpward(A, x, y, neighbors, count):

    N = size(A) // Size of the input n x n Matrix.
    count:= 0

    if y - 1 >= 0 then
        neighbors[count] := x * N + (y - 1)
        count:= count + 1
    endif

    if x - 1 >= 0 then
        neighbors[count] := (x - 1) * N + y
        count:= count + 1
    endif

end function

/**
* This is a helper function. It percolates down the Matrix.
*/
function percolateDownward(A, x, y, neighbors, count):

    N = size(A) // Size of the input n x n Matrix.
    count:= 0

    if y + 1 < N then
        neighbors[count] := x * N + (y + 1)
        count:= count + 1
    endif

    if x + 1 < N then
        neighbors[count] := (x + 1) * N + y
        count:= count + 1
```

```
    endif

end function

/**
* This is a helper function. It adjusts up the matrix.
*/
function adjustUpward(A, x, y, newX, newY)

    N = size(A) // Size of the input n x n Matrix.
    neighbors := array of size 2
    count := 0
    nX := 0
    nY := 0
    min_idx := -1

    percolateUpward(A, x, y, neighbors, count)

    if count == 0 then
        newX := -1
        newY := -1
        return
    endif

    min_val := INF

    for k := 0 to count - 1 do

        nX := neighbors[k] / N
        nY := neighbors[k] % N

        if A[nX][nY] < min_val then
            min_val := A[nX][nY]
            min_idx := k
        endif

    endfor

    if min_idx != -1 and A[x][y] < min_val then
```

```
        nX := neighbors[min_idx] / N
        nY := neighbors[min_idx] % N
        temp := A[x][y]
        A[x][y] := A[nX][nY]
        A[nX][nY] := temp
        newX := nX
        newY := nY

    else
        newX := -1
        newY := -1
    endif

end function

/**
* This is a helper function. It adjusts down the matrix.
*/
function adjustDownward(A, x, y, newX, newY)

    N = size(A) // Size of the input n x n Matrix.
    neighbors := array of size 2
    count := 0
    nX := 0
    nY := 0
    min_idx := -1

    percolateDownward(A, x, y, neighbors, count)

    if count == 0 then
        newX := -1
        newY := -1
        return
    endif

    min_val := A[x][y]

    for k := 0 to count - 1 do

        nX := neighbors[k] / N
```

```
        nY := neighbors[k] % N

        if A[nX][nY] < min_val then
            min_val := A[nX][nY]
            min_idx := k
        endif

    endfor

    if min_idx != -1 then

        nX := neighbors[min_idx] / N
        nY := neighbors[min_idx] % N
        temp := A[x][y]
        A[x][y] := A[nX][nY]
        A[nX][nY] := temp
        newX := nX
        newY := nY
    else
        newX := -1
        newY := -1
    endif

end function
```

Part 1: Let us now implement the **popmin(A, n)** function.

```
/**
* Pop minimum value in the Matrix.
*/
function popmin(A, n):

    if n == 0 then
        return -1 // No element to pop from an empty matrix
    endif

    // The smallest element is at the top-left corner (0, 0)
    element := A[0][0]

    // Set the smallest element to INF
    A[0][0] := INF

    // Start moving down from the top-left corner
    x := 0
    y := 0
    newX := 0
    newY := 0
    adjustDownward(A, x, y, newX, newY)

    // Continue adjusting downward until no more downward adjustment is
possible
    while newX != -1 and newY != -1 do
        x := newX
        y := newY
        adjustDownward(A, x, y, newX, newY)
    endwhile

    return element

end function
```

Part 2: Let us now implement **inser(A, n, value)** function.

```
/**
* Inserts a value into the Matrix.
*/
function insert(A, n, value)

    if n == 0 then
        return // Can't insert into an empty matrix
    endif

    // Place the value in the bottom-right corner
    A[n - 1][n - 1] := value

    // Start moving up from the bottom-right corner
    x := n - 1
    y := n - 1
    newX := 0
    newY := 0

    adjustUpward(A, x, y, newX, newY)

    // Continue adjusting upward until no more upward adjustment is
possible
    while newX != -1 and newY != -1 do
        x := newX
        y := newY
        adjustUpward(A, x, y, newX, newY)
    endwhile

end function
```

Part 3: Let us now implement a sorting algorithm for the matrix.

```
/**
* Matrix sorting algorithm.
*/
function sort(A, size)

    // Create a temporary new Matrix to hold sorted elements
    newMatrix := 2D array of size n x n // size x size
    temp := 0

    // Initialize the new matrix with INF values.
    for i := 0 to size - 1 do
        for j := 0 to size - 1 do
            newMatrix[i][j] := INF
        endfor
    endfor

    // Insert all elements from A into the new matrix.
    for i := 0 to size - 1 do
        for j := 0 to size - 1 do
            insert(newMatrix, n, A[i][j])
        endfor
    endfor

    // Extract sorted elements back into A
    for i := 0 to size - 1 do
        for j := 0 to size - 1 do
            A[temp / size][temp % size] := popmin(newMatrix, size)
            temp := temp + 1
        endfor
    endfor

end function
```

In the developed sorting algorithm, we have used the algorithms that were developed in previous parts of this exercise. Specifically, we have used the **popmin()** and **insert()** functions. In our implementation, we have also used a new $n \times n$ matrix as described in the question.

In the first pair of loops where we are initializing the new matrix, we have a complexity of $O(n^2)$.

In the second pair of loops where we are inserting elements into the new matrix, we have a complexity of $O(n^3)$.

In the third pair of loops where we are extracting sorted elements, we have a complexity of $O(n^2)$.

To put everything together, we have an overall time complexity of:

$$
\begin{aligned}
f(n) &= n^2 + (n^2 + n) + n^2 \\
&= n^2 + n^3 + n^2 \\
&= n^3 + 2n^2
\end{aligned}
$$

We shall now prove that $f(n) \in O(n^3)$

Let c and m be some constants. Such that,
$n > m$

$f(n) \leq c * g(n)$, where $g(n) = n^3$ and $\forall n \geq m$

Let us choose $c = 3$ and $m = 1$

So we have,

$f(n) \leq c * g(n)$
$n^3 + 2n^2 \leq n^3 + 2n^3$
$n^3 + 2n^2 \leq 3n^3$

We can see that $f(n)$ is in $g(n)$ or in $O(n^3)$.

2a) Prove correctness of **partition**.

We can use a trace table for the first step (loop invariant).
Let us take a small input,

Say,

$A = [3, 8, 6, 1, 5]$

$low = 0$

$high = 5 - 1 = 4$

$pivot = A[high] = 5$

$i = 0 - 1 = -1$

Tracing Table in the loop:

i	j	Loop Guard ($j \leq high - 1$)	Swapping Process swap(A, i, j)
-1	0	$0 \leq 4$	[<u>3</u>, 8, 6, 1, 5]
0	1	$1 \leq 4$	N/A
0	2	$2 \leq 4$	N/A
1	3	$3 \leq 4$	[3, 1, 6, 8, 5]
1	4	$4 \leq 4$	N/A

By tracing the algorithm we can come up with a loop with an appropriate loop invariant. The loop invariant will consist of three parts.

Part 1 of the invariant:

We know that i and j variables are indices within the input array A bounds (between low to $high$).

The variable i is always less than j (we can see that from the trace table too). This ensures that there are elements between i and j that have not been compared yet.

So the first part of the loop invariant is the following:

$inv_1(i, j) :\ 0 \le i < j \le high$

Part 2 of the invariant:

All elements from the start of the array (low) up to i (inclusive) are less than or equal to the value of $pivot$.

The variable i keeps track of the last position where all of the elements before it are less than or equal to the value of $pivot$.

So, to express the second part of the invariant; let k represent an index in the input array A, where $k \in \mathbb{N}$

For all indices k such that $low \le k \le i$

So,

$inv_2(i, k)$: $\forall k \in [low,\ i]$: $A[k]\ \le pivot$

Part 3 of the invariant:

In this part of the invariant, we should guarantee that all elements between $i + 1$ and $j - 1$ (inclusive) are greater than or equal to the value of $pivot$.

The variable j represents the current position being considered for partitioning, and it has not been compared yet.

Let us use the same variable k from part 2,

For all indices k such that $i + 1 \le k < j$

$inv_3(i, j, k)$: $\forall k \in [i + 1, . . . , j - 1]$: $A[k]\ \ge pivot$

Now, let us combine the three parts of the invariant; the invariant can be expressed as:

$inv(i, j, k)$: $inv_1(i, j) \wedge inv_2(i, k) \wedge inv_3(i, j, k)$

$inv(i, j, k)$: $(0 \le i < j \le high) \wedge (\forall k \in [low,\ i]\colon A[k] \le pivot)$
$$\wedge\ \forall k \in [i + 1, \ldots, j - 1]\colon A[k] \ge pivot$$

Proof of Invariant:

Base Case:

We have the following values for the variables $i,\ j,\ low,\ high$, and $pivot$.

$i = low - 1$
$j = low$
$low = 0$
$high = size\ of\ A - 1$
$pivot = A[high]$

Before entering the loop, the first part of the invariant ($inv_1(i, j)$) holds.
$inv_1(i, j) : 0 \le i < j \le high$
$inv_1(low - 1,\ low) : 0 \le low - 1 < low \le high$

We can see that the inequality holds for $inv_1(i, j)$

There are no elements between low and i yet, so $inv_2(i, k)$ holds trivially.
There are no elements between $i + 1$ and $j - 1$ yet, so $inv_3(i, j, k)$ also holds trivially.

Inductive Hypothesis:

Let i_1 and j_1 be variables in some random iteration of the loop. Upto this iteration L the invariant holds. At this iteration L the values of the variables are i_1 and j_1

So,

$inv(i_1, j_1, k)$: $inv_1(i_1, j_1) \wedge inv_2(i_1, k) \wedge inv_3(i_1, j_1, k)$
$inv(i_1, j_1, k)$: $(0 \le i_1 < j_1 \le high) \wedge (\forall k \in [low,\ i_1]\colon A[k] \le pivot)$
$$\wedge\ \forall k \in [i_1 + 1, \ldots, j_1 - 1]\colon A[k] \ge pivot$$
Performing one more iteration. The following values of i and j are after the iteration $L + 1$ is complete. So we will have,

$i_2 = i_1 + 1$ and $j_2 = j_1 + 1$

$$inv(i_2, j_2, k): (0 \leq i_2 < j_2 \leq high) \wedge (\forall k \in [low,\ i_2]:\ A[k] \leq pivot)$$
$$\wedge\ \forall k \in [i_2 + 1, ..., j_2 - 1]:\ A[k] \geq pivot$$
$$inv(i_2, j_2, k): (0 \leq i_1 + 1 < j_1 + 1 \leq high) \wedge (\forall k \in [low,\ i_1 + 1]:\ A[k] \leq pivot)$$
$$\wedge\ \forall k \in [(i_1 + 1) + 1, ..., (j_1 + 1) - 1]:\ A[k] \geq pivot$$

Let us examine parts by parts of the invariant. Let us examine the first part of the invariant:

$$inv_1(i_2, j_2):\ 0 \leq i_2 < j_2 \leq high$$
$$inv_1(i_2, j_2):\ (0 \leq i_1 + 1 < j_1 + 1 \leq high)$$

This will always be true because the loop maintains the bounds $0 \leq i < j \leq high$. Maintaining the bounds throughout the execution of the algorithm ensures that the partitioning step correctly divides the array into two parts.

Let us examine the second part of the invariant:

For all indices k such that $low \leq k \leq i$

$$inv_2(i_2, k):\ \forall k \in [low,\ i_2]:\ A[k] \leq pivot$$
$$inv_2(i_2, k):\ \forall k \in [low,\ i_1 + 1]:\ A[k] \leq pivot$$

In the IF statement located in the body of the loop, a swap occurs with $A[j_1 + 1]$ and $A[i_1 + 1]$. This maintains $inv_2(i_2, k)$ because $A[i_1 + 1]$ was greater than or equal to the $pivot$ before the swap happened and $A[j_1 + 1]$ is now placed correct as $A[i_1 + 1]$, so $inv_2(i_2, k)$ holds throughout the loop.

Let us examine the third part of the invariant:
For all indices k such that $i + 1 \leq k < j$

$$inv_3(i_2, j_2, k):\ \forall k \in [i_2 + 1, ..., j_2 - 1]:\ A[k] \geq pivot$$

$inv_3(i_2, j_2, k)$: $\forall k \in [(i_1 + 1) + 1, . . . , (j_1 + 1) - 1]$: $A[k] \geq pivot$
$inv_3(i_2, j_2, k)$: $\forall k \in [i_1 + 2, . . . , j_1]$: $A[k] \geq pivot$

The $A[j] > pivot$, then i remains unchanged, and j increments to the next value. This maintains $inv_3(i_2, j_2, k)$ because $A[j]$ is already greater than or equal to the $pivot$ value, and i remains unchanged, so $inv_3(i_2, j_2, k)$ holds throughout the loop.

Proof of Termination:

When the loop guard is violated, j becomes equal to $high$,

$j = high$

The variant is:

$V = high - j$

The variant decreases with each iteration of the loop. This is done to ensure that the loop will terminate after a finite number of iterations.

On the first iteration, the value of the variant is the highest, and the value of the variant at the end of the iteration is going to be the lowest.

We prove the following: $V \in \mathbb{N}$

Before we enter the loop,

$i = low - 1$
$j = low$
$low = 0$
$high = size\ of\ A - 1$
$pivot = A[high]$

Initially,

$$
\begin{aligned}
V &= high - j \\
&= high - low \\
&= high - 0 \\
&= high
\end{aligned}
$$

Since low and $high$ are indices of the array A, and $low \leq j \leq high$,

$V \geq 0$

Thus, $V \in \mathbb{N}$ the highest value initially.

When we are in any iteration, that means we are passed the loop guard. The loop guard is $j \leq high - 1$

So,

$j \leq high - 1$
$\Rightarrow 0 \geq high - 1 - j$
$\Rightarrow high - 1 - j \geq 0$

During each iteration of the loop, j is incremented by 1, so this decreases V each time because $V = high - j$. Therefore, V decreases until the loop terminates.

To show $V = high - j$ decreases,

Let j_1 and be on an iteration. Let's perform one more iteration, we have:

$j_2 = j_1 + 1$

Checking the difference:

$(high - j_1) - (high - j_2)$
$= (high - j_1) - [high - (j_1 + 1)]$
$= (high - j_1) - (high - j_1 - 1)$
$= high - j_1 - high + j_1 + 1$
$= 1$

Since $1 > 0$, the function is strictly decreasing at each iteration of the loop. This shows that the code terminates.

The final swap ensures that $A[i + 1]$ is the pivot element and the code returns $i + 1$ (the correct position of the $pivot$) as expected.

2b) Use **partition** to write a version of **Quicksort** that sorts a numerical array A in place.

We will write the QuickSort algorithm in pseudocode.

```
/*
* PRE: A is an array of numbers. low and high are valid
*      indices of A such that 0 <= low <= high < size of A
*
* POST: The subarray A[low...high] is sorted in
*        non-descending order.
*/
QuickSort(A, low, high):

    if low < high then do:

        pivot = partition(A, low, high)

        QuickSort(A, low, pivot - 1)
        QuickSort(A, pivot + 1, high)
```

3)

Capturing the worst-case runtime for Method 1:

$T(n) = 7T\left(\frac{n}{7}\right) + 12n + 3$

$a = 7,\ b = 7,\ f(n) = 12n + 3$

$log_b a$

$= log_7(7)$

$= 1$

$f(n) = 12n + 3$

$\quad = (12n + 3)(log^p n)$

From here we can see that:
$k = 1,\ p = 0$

We can also see that:

$log_b a = k$

$1 = 1$

So this must be case #2.

Since $p > -1$

$T(n) = \Theta(n^k log^{p+1} n)$

So,

$T(n) = \Theta(n^1 log^{0+1} n)$

$T(n) = \Theta(nlogn)$

Capturing the worst-case runtime for Method 2:

$T(n) = T(n - 2) + 120$

Given the format of this recurrence, we cannot apply Master's Theorem. We have to solve this recurrence by using another method. Let us use the Iterative method.

<u>Base case</u>: Let us assume that it takes a constant amount of time, $T(0) = c$, where c is just some constant.

We can start with the original $T(n)$

$T(n) = T(n - 2) + 120$

Substitute $n - 2$:

So,

$T(n - 2) = T[(n - 2) - 2] + 120$

$\quad = T(n - 4) + 120$

So,

$T(n) = [T(n - 4) + 120] + 120$

$\quad = T(n - 4) + 2 * 120$

Substitute $n - 4$:

$T(n - 4) = T(n - 6) + 120$

So,

$T(n) = [T(n - 6) + 120] + 2 * 120$
$\quad = T(n - 6) + 3 * 120$
. . .

After continuing this process, we have:
$T(n) = T(n - 2k) + 120k$, where k is the number of steps.

Let us now find the value of k when $T(n - 2k)$ reaches the base case.

Let $n - 2k = 0$ and solving for k
$n = 2k$
$\frac{n}{2} = k$

Substituting k back into the equation:
$k = \frac{n}{2}$
$T(n) = T\left[n - 2\left(\frac{n}{2}\right)\right] + (120)\left(\frac{n}{2}\right)$
$\quad = T(0) + 60n$ Base case has it that $T(0) = c$, where c is a constant.
$\quad = c + 60n$

When n becomes large, the constant c becomes irrelevant. Thus, the recurrence simplifies to $T(n) = O(n)$

Capturing the worst-case runtime for Method 3:

$T(n) = 2T\left(\frac{n}{2}\right) + 2n^2 + 3n + 5$

$a = 2,\ b = 2,\ f(n) = 2n^2 + 3n + 5$

$log_b a$
$= log_2(2)$
$= 1$

$f(n) = 2n^2 + 3n + 5$

$= (2n^2 + 3n + 5)(log^p n)$

From here we can see that:
$k = 2,\ p = 0$

We can also see that:
$log_b a < k$

$1 < 2$

So this must be case #3
$T(n) = \Theta(n^k log^p n)$

$T(n) = \Theta(n^2 log^0 n)$

$= \Theta(n^2)$

Comparing the asymptotic runtimes:

Method 1: $\Theta(nlogn)$
Method 2: $\Theta(n)$
Method 3: $\Theta(n^2)$

Out of the three methods, **method #2** yields the fastest asymptotic runtime.

4 i) The following pseudocode shows the implementation of the “multiply_monotone()” function with the time complexity of $\Theta(n)$. In the next part of the question, we argue that the time complexity is indeed $\Theta(n)$.

```
/**
*   PRE: x and y are strings representing monotone numbers.
*   POST: Returns the product of x and y monotone numbers.
*/
function multiply_monotone(x, y):

    // Convert strings x and y to integers
    xInt = convert_to_integer(x)
    yInt = convert_to_integer(y)

    // Base case: if yInt or xInt is a single digit, perform direct
multiplication
    if length(y) == 1 or length(x) == 1:
        return xInt * yInt

    // Divide
    mid = length(y) / 2
    h = substring(y, 0, mid)

    recursiveResult = multiply_monotone(x, h)

    string1 = convert_to_string(recursiveResult)
    string2 = string1 + "0" * mid
    finalProduct = convert_to_integer(string2)

    // Conquer
    return finalProduct + recursiveResult
```

ii) From the previous part, we can see there are multiple function calls. However, there is only one recursive call to the monotone function. Note that in that recursive call, we are inputting x and h, where h is basically half of the input size.

Since we are halving the input size in the recursive call, $b = 2$, and near the end, it will take linear time to combine the results from the recursive call.

So, our recurrence turns out to be:

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

We can use the Master Theorem to argue that the time complexity of the "multiply_montone" function is $\Theta(n)$

So,

$T(n) = T\left(\frac{n}{2}\right) + n$
$a = 1,\ b = 2,\ f(n) = n$

$log_b a$
$= log_2(1)$
$= 0$

Using the form on $f(n)$: $f(n) = n^k log^p(n)$

In our case, $f(n) = n$
So, $k = 1,\ p = 0$

We can see that:
$log_b a < k$
$0 < 1$

So, this must be case #3. Since $p \geq 0$, we have the following:
$T(n) = \Theta(n^k log^p n)$
$= \Theta(n^1 log^0 n)$
$= \Theta(n)$

After using the Master Theorem, we have ended up with $\Theta(n)$ as the time complexity for the "multiply_monotone" function.