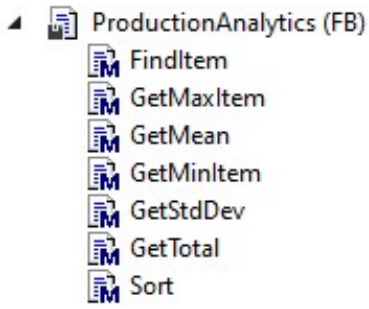
Production Data Analytics

This Function Block evaluates many useful statistics on production dataset (array of REAL data) using algorhitms like Binary search and Heapsort.

It determines following results with usage of dedicated `METHODs`:

- Mean value of data set (`GetMean`)
- Standard deviation od data set (`GetStdDev`)
- Total sum of all items (`GetTotal`)
- Check if specified item exists in data set (using Binary search) and get its index position (`FindItem`)
- Sort all items in data set using Heapsort algorithm (`Sort`)
- Determine maximum value of data set (`GetMax`)
- Determine minumum value of data set (`GetMin`)




```
FUNCTION_BLOCK ProductionAnalytics

VAR_IN_OUT
	// Input array for analytics
	DataArray		: ARRAY[1..ARRAY_SIZE] OF REAL;
END_VAR
VAR
	Idx			: INT;		// loop index
END_VAR

VAR CONSTANT
	ARRAY_SIZE		: UINT := 10;
END_VAR
```

```
METHOD GetMean : REAL
VAR_INPUT
END_VAR

VAR
	Sum		: REAL;		// Sum of array elements
	MeanTemp	: REAL;		// Temporary mean value for calculations
	Idx		: UINT;		// Loop counter
END_VAR

(* Calculate the Mean *)
FOR Idx := 1 TO ARRAY_SIZE DO
	Sum := Sum + THIS^.DataArray[Idx];
END_FOR;


MeanTemp := Sum / TO_REAL(ARRAY_SIZE);
GetMean := MeanTemp;
```

```
METHOD GetStdDev : REAL
VAR_INPUT
END_VAR

VAR
	MeanTemp		: REAL;		// Temporary mean value for calculations
	Deviation		: REAL;		// Deviation of each element from the mean
	SumOfSquares		: REAL;		// Sum of squared differences from the mean
	Idx			: UINT;		// Loop counter
	ArrayElement		: REAL;		// Casted array element to REAL for precision
	MeanError		: BOOL;
END_VAR

(* Calculate the Standard Deviation *)
MeanTemp := THIS^.GetMean();

IF ARRAY_SIZE > 0 THEN
	FOR Idx := 1 TO ARRAY_SIZE DO
		ArrayElement := THIS^.DataArray[Idx];
		Deviation := ArrayElement - MeanTemp;
		SumOfSquares := SumOfSquares + (Deviation * Deviation);
	END_FOR;

	GetStdDev := SQRT(SumOfSquares / TO_REAL(ARRAY_SIZE));
END_IF;
```

```
METHOD GetTotal : REAL
VAR_INPUT
END_VAR

VAR
	Idx			: UINT;			// Loop counter
	ArrayElement		: REAL;			// Temporary array value
	Total			: REAL;
END_VAR

FOR Idx := 1 TO ARRAY_SIZE DO
	ArrayElement := DataArray[Idx];
	Total := Total + ArrayElement;
END_FOR;

GetTotal := Total;
```

```
METHOD FindItem
VAR_INPUT
    SearchValue : REAL;             // The value to search for
    StartIndex  : UINT;             // Start index of the array to search in (1-based)
    EndIndex    : UINT;             // End index of the array to search in (1-based)
END_VAR

VAR_OUTPUT
    FoundIndex  : UINT;             // Index of the found value (1-based), 0 if not found
    Found       : BOOL;             // TRUE if the value is found, FALSE otherwise
    ValidInput  : BOOL;             // Indicates if the input parameters are valid
END_VAR

VAR
    MidIndex    : UINT;              // Midpoint index for binary search
    Low         : UINT;              // Lower boundary for search
    High        : UINT;              // Upper boundary for search
END_VAR

(* Find element using binary search algorithm *)

// Input validation
IF (StartIndex >= 1) AND (EndIndex <= ARRAY_SIZE) AND (StartIndex <= EndIndex) THEN
    ValidInput := TRUE;
ELSE
    ValidInput := FALSE;
    Found := FALSE;
    FoundIndex := 0;
    RETURN;
END_IF;

// Binary Search Algorithm
Low := StartIndex;
High := EndIndex;
Found := FALSE;

WHILE (Low <= High) AND (NOT Found) DO
    // Calculate midpoint
    MidIndex := (Low + High) / 2;

    IF DataArray[MidIndex] = SearchValue THEN
        // Value found
        Found := TRUE;
        FoundIndex := MidIndex;
    ELSIF DataArray[MidIndex] < SearchValue THEN
        // Search in the right half
        Low := MidIndex + 1;
    ELSE
        // Search in the left half
        High := MidIndex - 1;
    END_IF;
END_WHILE;

// If not found, set FoundIndex to 0
IF NOT Found THEN
    FoundIndex := 0;  // Set to 0 to indicate not found
END_IF;
```

```
METHOD Sort
VAR_INPUT
END_VAR

VAR_OUTPUT
    SortedArray   : ARRAY[1..ARRAY_SIZE] OF REAL;   // Output array with sorted elements
    IsSorted      : BOOL;                   // Indicates if the array is sorted
    ValidInput    : BOOL;                   // Indicates if the input parameters are valid
END_VAR

VAR
    Idx           : UINT;                   // Loop indices
    Parent, Child : UINT;                   // Indices for heap construction
    Temp          : REAL;                   // Temporary variable for swapping
END_VAR

(* Sort items using Heapsort algorithm*)

// --- Input Validation ---
IF (ARRAY_SIZE < 1) OR (ARRAY_SIZE > 100) THEN
    ValidInput := FALSE;
    IsSorted := FALSE;
    RETURN;
ELSE
    ValidInput := TRUE;
END_IF;

// --- Copy input array to output array ---
FOR Idx := 1 TO ARRAY_SIZE DO
    SortedArray[Idx] := THIS^.DataArray[Idx];
END_FOR;

// --- Heap Construction Phase ---
Idx := ARRAY_SIZE / 2;
WHILE Idx >= 1 DO
    Parent := Idx;
    WHILE 2 * Parent <= ARRAY_SIZE DO
        Child := 2 * Parent;

        // Select the larger child
        IF (Child < ARRAY_SIZE) AND
                    (SortedArray[Child] < SortedArray[Child + 1]) THEN
            Child := Child + 1;
        END_IF;

        // If parent is smaller than the largest child, swap them
        IF SortedArray[Parent] < SortedArray[Child] THEN
            Temp := SortedArray[Parent];
            SortedArray[Parent] := SortedArray[Child];
            SortedArray[Child] := Temp;

            // Move down the heap
            Parent := Child;
        ELSE
            EXIT;  // Break loop if the heap property is satisfied
        END_IF;
    END_WHILE;
```

```
    Idx := Idx - 1;  // Equivalent to DOWNTO without using DOWNTO
END_WHILE;

// --- Sorting Phase ---
FOR Idx := ARRAY_SIZE TO 2 BY -1 DO
    // Swap the first element (largest) with the last unsorted element
    Temp := SortedArray[1];
    SortedArray[1] := SortedArray[Idx];
    SortedArray[Idx] := Temp;

    // Restore the heap property for the reduced heap
    Parent := 1;
    WHILE 2 * Parent < Idx DO
        Child := 2 * Parent;

        // Select the larger child
        IF (Child < (Idx - 1)) AND (SortedArray[Child] < SortedArray[Child + 1]) THEN
            Child := Child + 1;
        END_IF;

        // If parent is smaller than the largest child, swap them
        IF SortedArray[Parent] < SortedArray[Child] THEN
            Temp := SortedArray[Parent];
            SortedArray[Parent] := SortedArray[Child];
            SortedArray[Child] := Temp;

            // Move down the heap
            Parent := Child;
        ELSE
            EXIT;  // Break loop if the heap property is satisfied
        END_IF;
    END_WHILE;
END_FOR;

// --- Set output signal to indicate sorting completion ---
IsSorted := TRUE;
```

```
METHOD GetMaxItem : REAL
VAR_INPUT
END_VAR

VAR
	ValidInput		: BOOL;
	SortedArray		: ARRAY[1..ARRAY_SIZE] OF REAL;
	IsSorted		: BOOL;
END_VAR

THIS^.Sort(SortedArray=>SortedArray,
		IsSorted=>IsSorted,
		ValidInput=>ValidInput);

If IsSorted then
	GetMaxItem := SortedArray[ARRAY_SIZE];
END_IF



METHOD GetMinItem : REAL
VAR_INPUT
END_VAR

VAR
	ValidInput		: BOOL;
	SortedArray		: ARRAY[1..ARRAY_SIZE] OF REAL;
	IsSorted		: BOOL;
END_VAR

THIS^.Sort(SortedArray=>SortedArray,
		IsSorted=>IsSorted,
		ValidInput=>ValidInput);

If IsSorted then
	GetMinItem := SortedArray[1];
END_IF
```