Array Utils Documentation

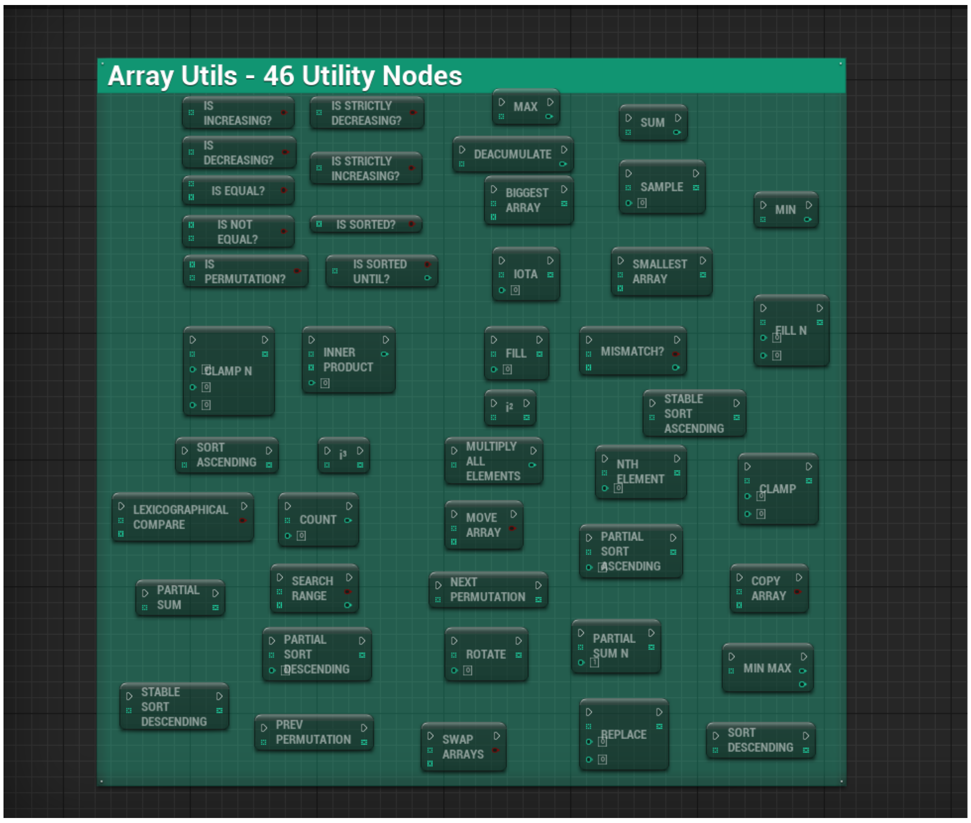

Figure 1: Array Utils Overview.

Introduction

This document provides comprehensive documentation for the `UNumericBPLibrary` class in Unreal Engine. The class contains a collection of powerful array utility functions designed to enhance numerical array manipulation.

Functions

ArrayMax

```
/**
 * Returns the maximum value of an integer array.
 *
 * @param A The input integer array.
 * @return The maximum value in the array.
```

a	r	r	a	y

```
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static int32 ArrayMax(const TArray<int32>& A);
```

Listing 1: Returns the maximum value of an array.

ArrayMin

```
/**
 * Returns the minimum value of an array.
 *
 * @param A The input array.
 * @return The minimum value in the array.
 */
UFUNCTION(BlueprintCallable,
            meta = (CompactNodeTitle = "MIN",
                    Category = "Array Utils",
                    ToolTip = "Returns the minimum value of an array"))
static int32 ArrayMin(const TArray<int32>& A);
```

Listing 2: Returns the minimum value of an array.

ArrayMinMax

```
/**
 * Returns the minimum and maximum value of an array.
 *
 * @param A The input array.
 * @param Min (Out) The minimum value in the array.
 * @param Max (Out) The maximum value in the array.
 */
UFUNCTION(BlueprintCallable,
            meta = (CompactNodeTitle = "MIN MAX",
                    Category = "Array Utils",
                    ToolTip = "Returns the minimum and maximum value of an
    array"))
static void ArrayMinMax(const TArray<int32>& A, int32& Min, int32& Max);
```

Listing 3: Returns the minimum and maximum value of an array.

PartialSum

```
/**
 * Returns the partial summed array. Example: partialsum({1,2,3,4}) ->
   {1,3,6,10}
 *
 * @param A The input integer array.
 * @return The array of partial sums.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> PartialSum(const TArray<int32>& A);
```

Listing 4: Returns the partial summed array.

PartialSumN

```
/**
 * Returns the partial summed array up until the index given.
 *
 * @param A The input array.
```

```cpp
 * @param N The index up until which to sum the array.
 * @return The partial summed array up until the index given.
 * @note partial sum n({1,2,3}, 2) -> {1,3,3}
 */
UFUNCTION(BlueprintCallable,
            meta = (CompactNodeTitle = "PARTIAL SUM N",
                    Category = "Array Utils",
                    ToolTip = "Returns the partial summed array up until the
    index provided. Example: partialsumn({1,2,3}, 2) -> {1,3,3}"))
static TArray<int32> PartialSumN(const TArray<int32>& A, int32 N = 1);
```
Listing 5: Returns the partial summed array up until the index given.

Clamp

```cpp
/**
 * Returns the array with all elements clamped to the specified range.
 *
 * @param A The input integer array.
 * @param Min The minimum clamping value.
 * @param Max The maximum clamping value.
 * @return The clamped array.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> Clamp(const TArray<int32>& A, int32 Min, int32 Max);
```
Listing 6: Returns the array with all elements clamped to the specified range.

ClampN

```cpp
/**
 * Returns the array with N first elements clamped to the specified range.
 *
 * @param A The input integer array.
 * @param Min The minimum clamping value.
 * @param Max The maximum clamping value.
 * @param N The number of elements to clamp.
 * @return The clamped array.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> ClampN(const TArray<int32>& A, int32 Min, int32 Max, int32
    N);
```
Listing 7: Returns the array with N first elements clamped to the specified range.

EveryoneSquared

```cpp
/**
 * Transforms each number in the array to be the square of itself.
 *
 * @param A The input integer array.
 * @return The array with each element squared.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> EveryoneSquared(const TArray<int32>& A);
```
Listing 8: Transforms each number to be the square of itself.

EveryoneCubed

```cpp
/**
 * Transforms each number in the array to be the cube of itself.
 *
 * @param A The input integer array.
 * @return The array with each element cubed.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> EveryoneCubed(const TArray<int32>& A);
```

Listing 9: Transforms each number to be the cube of itself.

StableSortAscending

```cpp
/**
 * Sorts the array in ascending order, preserving the relative order of
   elements with equivalent values.
 *
 * @param A The input integer array.
 * @return The sorted array in ascending order.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> StableSortAscending(const TArray<int32>& A);
```

Listing 10: Sorts the array in ascending order, preserving the relative order of elements with equivalent values.

StableSortDescending

```cpp
/**
 * Sorts the array in descending order, preserving the relative order of
   elements with equivalent values.
 *
 * @param A The input integer array.
 * @return The sorted array in descending order.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> StableSortDescending(const TArray<int32>& A);
```

Listing 11: Sorts the array in descending order, preserving the relative order of elements with equivalent values.

IsPermutation

```cpp
/**
 * Returns true if the two arrays are permutations of each other.
 *
 * @param A The first array.
 * @param B The second array.
 * @return true if the arrays are permutations, false otherwise.
 */
UFUNCTION(BlueprintCallable,
            BlueprintPure,
            meta = (CompactNodeTitle = "IS PERMUTATION?",
                    Category = "Array Utils",
                    ToolTip = "Returns true if the two arrays are permutations
   of each other"))
static bool IsPermutation(const TArray<int32>& A, const TArray<int32>& B);
```

Listing 12: Returns true if the two arrays are permutations of each other.

NextPermutation

```cpp
/**
 * Returns the next permutation of an array.
 *
 * @param A The input array.
 * @return The next permutation of the array.
 */
UFUNCTION(BlueprintCallable,
            meta = (CompactNodeTitle = "NEXT PERMUTATION",
                    Category = "Array Utils",
                    ToolTip = "Returns the next permutation of an array"))
static TArray<int32> NextPermutation(TArray<int32> A);
```

Listing 13: Returns the next permutation of an array.

PrevPermutation

```cpp
/**
 * Returns the previous permutation of an array.
 *
 * @param A The input array.
 * @return The previous permutation of the array.
 */
UFUNCTION(BlueprintCallable,
            meta = (CompactNodeTitle = "PREV PERMUTATION",
                    Category = "Array Utils",
                    ToolTip = "Returns the previous permutation of an array"))
static TArray<int32> PrevPermutation(TArray<int32> A);
```

Listing 14: Returns the previous permutation of an array.

IsSorted

```cpp
/**
 * Returns true if the array is sorted in ascending order.
 * For example, {1, 2, 3, 4} is sorted, but {1, 3, 2, 4} is not.
 *
 * @param A The input array.
 * @return true if the array is sorted, false otherwise.
 */
UFUNCTION(BlueprintCallable,
            BlueprintPure,
            meta = (CompactNodeTitle = "IS SORTED?",
                    Category = "Array Utils",
                    ToolTip = "Returns true if the array is sorted in ascending
    order."))
static bool IsSorted(const TArray<int32>& A);
```

Listing 15: Returns true if the array is sorted in ascending order.

InnerProduct

```cpp
/**
 * Returns the inner product of two arrays. Arrays must have the same length,
   otherwise 0 will be returned.
 *
 * @param A The first array.
 * @param B The second array.
 * @param StartIndex The initial value for the inner product calculation.
 * @return The inner product of the two arrays.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "INNER PRODUCT",
    Category = "Array Utils", ToolTip = "Returns the inner product of two arrays
    . Arrays must have the same length, otherwise -1 will be returned."))
static int32 InnerProduct(const TArray<int32>& A, const TArray<int32>& B, int32
     StartIndex = 0);
```

Listing 16: Returns the inner product of two arrays.

Count

```
/**
 * Returns the number of elements in the array that are equal to the specified
    value.
 *
 * @param A The input array.
 * @param Value The value to count in the array.
 * @return The count of elements equal to the specified value in the array.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "COUNT", Category = "
    Array Utils", ToolTip = "Returns the number of elements in the array that
    are equal to the specified value"))
static int32 Count(const TArray<int32>& A, int32 Value);
```

Listing 17: Returns the number of elements in the array that are equal to the specified value.

Accumulate

```
/**
 * Returns the sum of all elements of an array.
 *
 * @param A The input array.
 * @return The sum of all elements in the array.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "SUM", Category = "
    Array Utils", ToolTip = "Returns the sum of all elements of an array"))
static int32 Accumulate(const TArray<int32>& A);
```

Listing 18: Returns the sum of all elements of an array.

Fill

```
/**
 * Fills the array with a specified value.
 *
 * @param A The input array.
 * @param Value The value to fill the array with.
 * @return The array filled with the specified value.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "FILL", Category = "
    Array Utils", ToolTip = "Fills all the array with a number of choice"))
static TArray<int32> Fill(const TArray<int32>& A, int32 Value);
```

Listing 19: Fills the array with a specified value.

FillN

```
/**
 * Fills the array with a number of choice up to the provided number.
 *
 * @param A The input array.
 * @param Value The value to fill the array with.
 * @param N The number of elements to fill with the specified value.
 * @return The array filled with the specified value.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "FILL N", Category = "
    Array Utils", ToolTip = "Fills the array with a number of choice up to the
    provided number. Example: {1,2,3,4,5} filln(5, 3) = {5, 5, 5, 4, 5}"))
static TArray<int32> FillN(const TArray<int32>& A, int32 Value, int32 N);
```

Listing 20: Fills the array with a number of choice up to the provided number.

SortAscending

```
/**
 * Sorts the array in ascending order.
 *
 * @param A The input array.
 * @return The array sorted in ascending order.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "SORT ASCENDING",
    Category = "Array Utils", ToolTip = "Sorts the array in ascending order"))
static TArray<int32> SortAscending(const TArray<int32>& A);
```

Listing 21: Sorts the array in ascending order.

SortDescending

```
/**
 * Sorts the array in descending order.
 *
 * @param A The input array.
 * @return The array sorted in descending order.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "SORT DESCENDING",
    Category = "Array Utils", ToolTip = "Sorts the array in descending order"))
static TArray<int32> SortDescending(const TArray<int32>& A);
```

Listing 22: Sorts the array in descending order.

PartialSortAscending

```
/**
 * Sorts the array in ascending order up to the specified index.
 *
 * @param A The input array.
 * @param N The index up to which to sort the array.
 * @return The array sorted in ascending order up to the specified index.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "PARTIAL SORT ASCENDING
    ", Category = "Array Utils", ToolTip = "Sorts the array in ascending order
    up to the specified index, leaving the rest in unspecified order"))
static TArray<int32> PartialSortAscending(const TArray<int32>& A, int32 N);
```

Listing 23: Sorts the array in ascending order up to the specified index.

PartialSortDescending

```
/**
 * Sorts the array in descending order up to the specified index.
 *
 * @param A The input array.
 * @param N The index up to which to sort the array.
 * @return The array sorted in descending order up to the specified index.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "PARTIAL SORT
    DESCENDING", Category = "Array Utils", ToolTip = "Sorts the array in
    descending order up to the specified index, leaving the rest in unspecified
    order"))
static TArray<int32> PartialSortDescending(const TArray<int32>& A, int32 N);
```

Listing 24: Sorts the array in descending order up to the specified index.

IsSortedUntil

```
/**
 * Returns the index of the first element in the array that is not sorted.
 *
 * @param A The input array.
 * @param IsSorted (Out) True if the array is sorted, false otherwise.
 * @return The index of the first element in the array that is not sorted.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, meta = (CompactNodeTitle = "IS
    SORTED UNTIL?", Category = "Array Utils", ToolTip = "Returns the index of
    the first element in the array that is not sorted"))
static int32 IsSortedUntil(const TArray<int32>& A, bool& IsSorted);
```

Listing 25: Returns the index of the first element in the array that is not sorted.

Deaccumulate

```
/**
 * Returns the subtraction of all elements of an array, starting from 0.
 * Example: Deaccumulate({1, 2, 3}) -> -6. Flips the sign and +/- respectively.
 *
 * @param A The input array.
 * @return The subtraction of all elements of an array.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "DEACUMULATE", Category
     = "Array Utils", ToolTip = "Returns the subtraction of all elements of an
    array, starting from 0. {1,2,3} -> -6. Flips the sign and + or -
    respectively."))
static int32 Deaccumulate(const TArray<int32>& A);
```

Listing 26: Returns the subtraction of all elements of an array.

MultiplyAllElements

```
/**
 * Returns the multiplication of all elements of an array.
 *
 * @param A The input array.
 * @return The multiplication of all elements of an array.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "MULTIPLY ALL ELEMENTS"
    , Category = "Array Utils", ToolTip = "Returns the multiplication of all
    elements of an array"))
static int32 MultiplyAllElements(const TArray<int32>& A);
```

Listing 27: Returns the multiplication of all elements of an array.

LexicographicalCompare

```
/**
 * Returns true if the first array is lexicographically less than the second
   array.
 * For example, {1, 2, 3} is less than {1, 2, 4} as 3 < 4.
 *
 * @param A The first array.
 * @param B The second array.
 * @return True if the first array is lexicographically less than the second
   array, false otherwise.
 */
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle = "LEXICOGRAPHICAL
    COMPARE", Category = "Array Utils", ToolTip = "Returns true if the first
    array is lexicographically less than the second array. For example, {1, 2,
    3} is less than {1, 2, 4} as 3 < 4."))
static bool LexicographicalCompare(const TArray<int32>& A, const TArray<int32>&
     B);
```

Listing 28: Returns true if the first array is lexicographically less than the second array.

BiggestArray

```
/**
 * Returns the greater array of two summed arrays.
 *
 * @param A The first array.
 * @param B The second array.
 * @return The greater array of two summed.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
     = "BIGGEST ARRAY", ToolTip = "Returns the greater array"))
static TArray<int32> BiggestArray(const TArray<int32>& A, const TArray<int32>&
    B);
```

Listing 29: Returns the greater array of two summed arrays.

SmallestArray

```
/**
 * Returns the smallest array of two summed arrays.
 *
 * @param A The first array.
 * @param B The second array.
 * @return The smallest array of two summed.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
     = "SMALLEST ARRAY", ToolTip = "Returns the smallest array"))
static TArray<int32> SmallestArray(const TArray<int32>& A, const TArray<int32>&
     B);
```

Listing 30: Returns the smallest array of two summed arrays.

Iota

```
/**
 * Fills the array with a number that increments by 1 each index.
 *
 * @param A The input array.
 * @param Value The starting value for filling the array.
```

```cpp
 * @return The array filled with values starting from the specified value and
   incrementing by 1.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
    = "IOTA", ToolTip = "Fills the array with a number that increments by 1
   each index, ex. iota({1,2,3}, 5) = {5,6,7}"))
static TArray<int32> Iota(const TArray<int32>& A, int32 Value);
```

Listing 31: Fills the array with a number that increments by 1 each index.

Replace

```cpp
/**
 * Replaces all instances of a value in an array with another value.
 *
 * @param A The input array.
 * @param OldValue The value to be replaced.
 * @param NewValue The new value to replace the old value with.
 * @return The array with replaced values.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
    = "REPLACE", ToolTip = "Replaces all instances of a value in an array with
   another value"))
static TArray<int32> Replace(const TArray<int32>& A, int32 OldValue, int32
   NewValue);
```

Listing 32: Replaces all instances of a value in an array with another value.

Rotate

```cpp
/**
 * Rotates the array by a specified amount.
 *
 * @param A The input array.
 * @param Amount The amount by which to rotate the array.
 * @return The rotated array.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
    = "ROTATE", ToolTip = "Rotates the array by a specified amount"))
static TArray<int32> Rotate(const TArray<int32>& A, int32 Amount);
```

Listing 33: Rotates the array by a specified amount. Positive values rotate to the right, negative values rotate to the left.

ArrayIsEqual

```cpp
/**
 * Returns true if the two arrays are equal.
 *
 * @param A The first array.
 * @param B The second array.
 * @return true if the two arrays are equal, false otherwise.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, Category = "Array Utils", meta = (
   CompactNodeTitle = "IS EQUAL?", ToolTip = "Returns true if the two arrays
   are equal"))
static bool ArrayIsEqual(const TArray<int32>& A, const TArray<int32>& B);
```

Listing 34: Returns true if the two arrays are equal.

ArrayIsNotEqual

```cpp
/**
 * Returns true if the two arrays are not equal.
 *
 * @param A The first array.
 * @param B The second array.
 * @return true if the two arrays are not equal, false otherwise.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, Category = "Array Utils", meta = (
    CompactNodeTitle = "IS NOT EQUAL?", ToolTip = "Returns true if the two
    arrays are not equal"))
static bool ArrayIsNotEqual(const TArray<int32>& A, const TArray<int32>& B);
```

Listing 35: Returns true if the two arrays are not equal.

NthElement

```cpp
/**
 * Returns the array with the nth element sorted.
 *
 * @param A The input array.
 * @param N The index of the element to sort.
 * @return The array with the nth element sorted.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
     = "NTH ELEMENT", ToolTip = "Returns the array with the nth element sorted")
    )
static TArray<int32> NthElement(const TArray<int32>& A, int32 N);
```

Listing 36: Returns the array with the nth element sorted.

Mismatch

```cpp
/**
 * Returns the index of the first mismatching element between two arrays.
 *
 * @param A The first array.
 * @param B The second array.
 * @param IsMismatch (Out) Whether the arrays are mismatched.
 * @return The index of the first mismatching element between two arrays.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils", meta = (CompactNodeTitle
     = "MISMATCH?", ToolTip = "Returns the index of the first mismatching
    element between two arrays"))
static int32 Mismatch(const TArray<int32>& A, const TArray<int32>& B, bool&
    IsMismatch);
```

Listing 37: Returns the index of the first mismatching element between two arrays. False if no mismatch (arrays are equal).

ArrayIsDecreasing

```cpp
/**
 * Returns true if the array is decreasing; i.e., each item is less than or
    equal to the previous one.
 *
 * @param A The input array.
 * @return true if the array is decreasing, false otherwise.
 */
```

```
UFUNCTION(BlueprintCallable, BlueprintPure, Category = "Array Utils", meta = (
    CompactNodeTitle = "IS DECREASING?", ToolTip = "Whether an entire sequence
    is decreasing; i.e., each item is less than or equal to the previous one"))
static bool ArrayIsDecreasing(const TArray<int32>& A);
```

Listing 38: Returns true if the array is decreasing.

ArrayIsIncreasing

```
/**
 * Returns true if the array is increasing; i.e., each item is greater than or
    equal to the previous one.
 *
 * @param A The input array.
 * @return true if the array is increasing, false otherwise.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, meta = (CompactNodeTitle = "IS
    INCREASING?", Category = "Array Utils", ToolTip = "Whether an entire
    sequence is increasing; i.e., each item is greater than or equal to the
    previous one"))
static bool ArrayIsIncreasing(const TArray<int32>& A);
```

Listing 39: Returns true if the array is increasing; i.e., each item is greater than or equal to the previous one.

ArrayIsStrictlyDecreasing

```
/**
 * Returns true if the array is strictly decreasing; i.e., each item is less
    than the previous one.
 *
 * @param A The input array.
 * @return true if the array is strictly decreasing, false otherwise.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, meta = (CompactNodeTitle = "IS
    STRICTLY DECREASING?", Category = "Array Utils", ToolTip = "Whether an
    entire sequence is strictly decreasing; i.e., each item is less than the
    previous one"))
static bool ArrayIsStrictlyDecreasing(const TArray<int32>& A);
```

Listing 40: Returns true if the array is strictly decreasing; i.e., each item is less than the previous one.

ArrayIsStrictlyIncreasing

```
/**
 * Returns true if the array is strictly increasing; i.e., each item is greater
     than the previous one.
 *
 * @param A The input array.
 * @return true if the array is strictly increasing, false otherwise.
 */
UFUNCTION(BlueprintCallable, BlueprintPure, meta = (CompactNodeTitle = "IS
    STRICTLY INCREASING?", Category = "Array Utils", ToolTip = "Whether an
    entire sequence is strictly increasing; i.e., each item is greater than the
    previous one"))
static bool ArrayIsStrictlyIncreasing(const TArray<int32>& A);
```

Listing 41: Returns true if the array is strictly increasing; i.e., each item is greater than the previous one.

EraseAllOccurrencesOfValue

```
/**
 * Erases all occurrences of a value in the array.
 *
 * @param A The input array.
 * @param Value The element to remove.
 * @return The array with the element(s) removed.
 */
UFUNCTION(Blueprintable, meta = (CompactNodeTitle = "ERASE FROM ARRAY",
    Category = "Array Utils", ToolTip = "Erases all occurrences of the value in
    the array. Ex Erase({1,2,3,4,5,5},5) -> {1,2,3,4}"))
static TArray<int32> EraseAllOccurrencesOfValue(UPARAM(ref) TArray<int32>& A,
    int32 Value);
```

Listing 42: Erases all occurrences of a value in the array.

ShrinkToFit

```
/**
 * Shrinks the array to fit the number of elements in the array.
 *
 * @param A The input array.
 * @return Shrink Array.
 */
 UFUNCTION(Blueprintable, meta = (CompactNodeTitle = "SHRINK TO FIT", Category
    = "Array Utils", ToolTip = "Shrinks the array to fit the number of elements
    in the array."))
 static TArray<int32> ShrinkToFit(UPARAM(ref) TArray<int32>& A);
```

Listing 43: Shrinks the array to fit the number of elements in the array.

CopyArray

```
/**
 * Copies the array A into B. Returns true if the copy was successful, false
    otherwise.
 *
 * @param A Array to copy.
 * @param B Array to copy into.
 * @return Whether the copy was successful.
 */
UFUNCTION(Blueprintable, BlueprintCallable, meta = (CompactNodeTitle = "COPY
    ARRAY", Category = "Array Utils", ToolTip = "Copies the array A into B.
    Returns true if the copy was successful, false otherwise."))
static bool CopyArray(const TArray<int32>& A, UPARAM(ref)TArray<int32>& B);
```

Listing 44: Copies the array A into B. Returns true if the copy was successful, false otherwise.

SwapArrays

```
/**
 * Swaps the contents of two arrays. Returns true if the swap was successful,
    false otherwise.
 *
 * @param A Array to swap.
 * @param B Array to swap.
 * @return Whether the swap was successful.
 */
UFUNCTION(Blueprintable, BlueprintCallable, meta = (CompactNodeTitle = "SWAP
    ARRAYS", Category = "Array Utils", ToolTip = "Swaps the contents of two
    arrays. Returns true if the swap was successful, false otherwise."))
```

```
static bool SwapArrays(UPARAM(ref) TArray<int32>& A, UPARAM(ref) TArray<int32>&
    B);
```

Listing 45: Swaps the contents of two arrays. Returns true if the swap was successful, false otherwise.

MoveArray

```
/**
 * Moves the contents of array A into B. Returns true if the move was
   successful, false otherwise.
 *
 * @param A Array to move.
 * @param B Array to move into.
 * @return Whether the move was successful.
 */
UFUNCTION(Blueprintable, BlueprintCallable, meta = (CompactNodeTitle = "MOVE
    ARRAY", Category = "Array Utils", ToolTip = "Moves the contents of array A
    into B. Returns true if the move was successful, false otherwise."))
static bool MoveArray(UPARAM(ref) TArray<int32>& A, UPARAM(ref) TArray<int32>&
    B);
```

Listing 46: Moves the contents of array A into B. Returns true if the move was successful, false otherwise.

Sample

```
/**
 * Returns N random numbers from the array. Example: sample({1,2,3,4,5}, 2) ->
   {3,2}
 *
 * @param A The input integer array.
 * @param N The number of random elements to sample.
 * @return The array of sampled elements.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static TArray<int32> Sample(const TArray<int32>& A, int32 N);
```

Listing 47: Returns N random numbers from the array.

Search

```
/**
 * Searches for the first occurrence of the sequence of elements in the first
   array.
 *
 * @param A The target array to search in.
 * @param B The array representing the sequence to search for.
 * @param found Output parameter indicating if the sequence was found.
 * @return The index of the first occurrence if found, -1 otherwise.
 */
UFUNCTION(BlueprintCallable, Category = "Array Utils")
static int32 Search(const TArray<int32>& A, const TArray<int32>& B, bool& found
    );
```

Listing 48: Searches for the first occurrence of the sequence of elements in the first array.

Conclusion

Thank you for reviewing the documentation for the `UNumericBPLibrary` class. We hope this comprehensive guide provides clarity on the functionality and proper usage of the various utility functions offered by the library. For any questions, ask in the marketplace. Happy blueprinting!