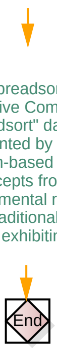
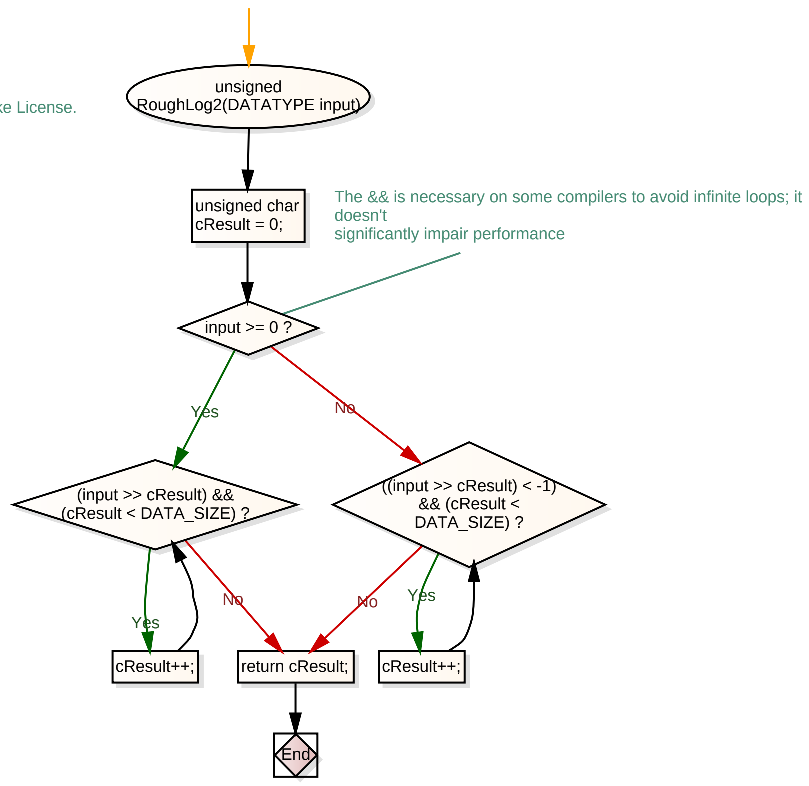
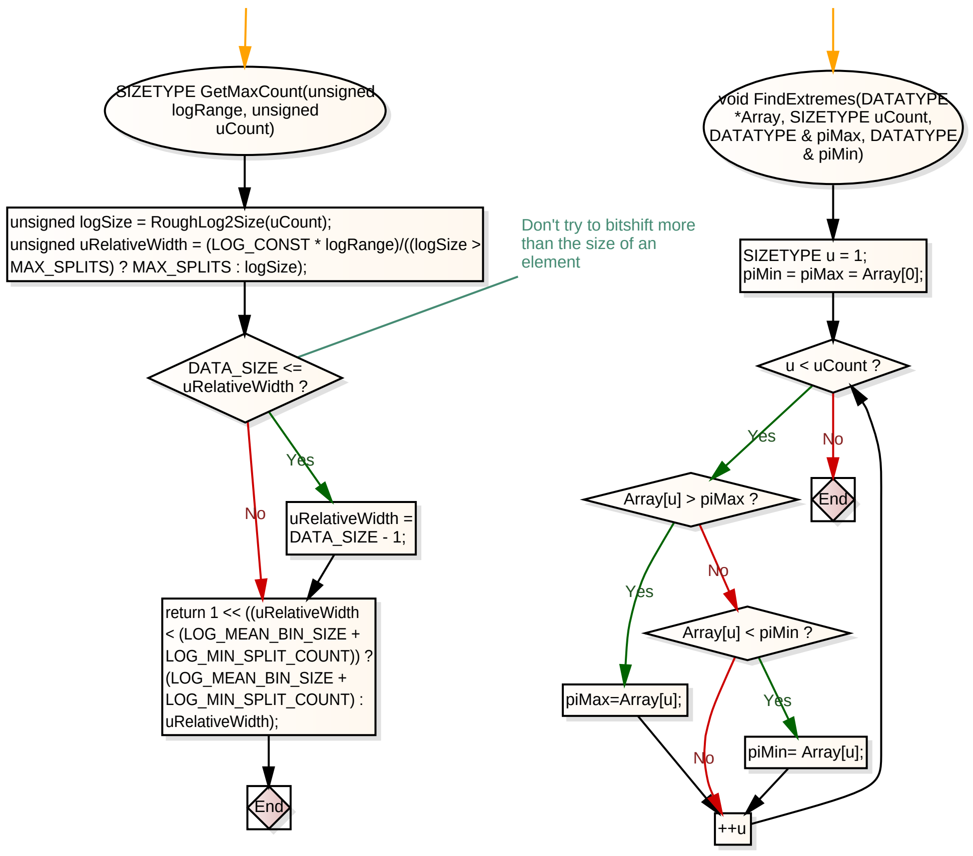
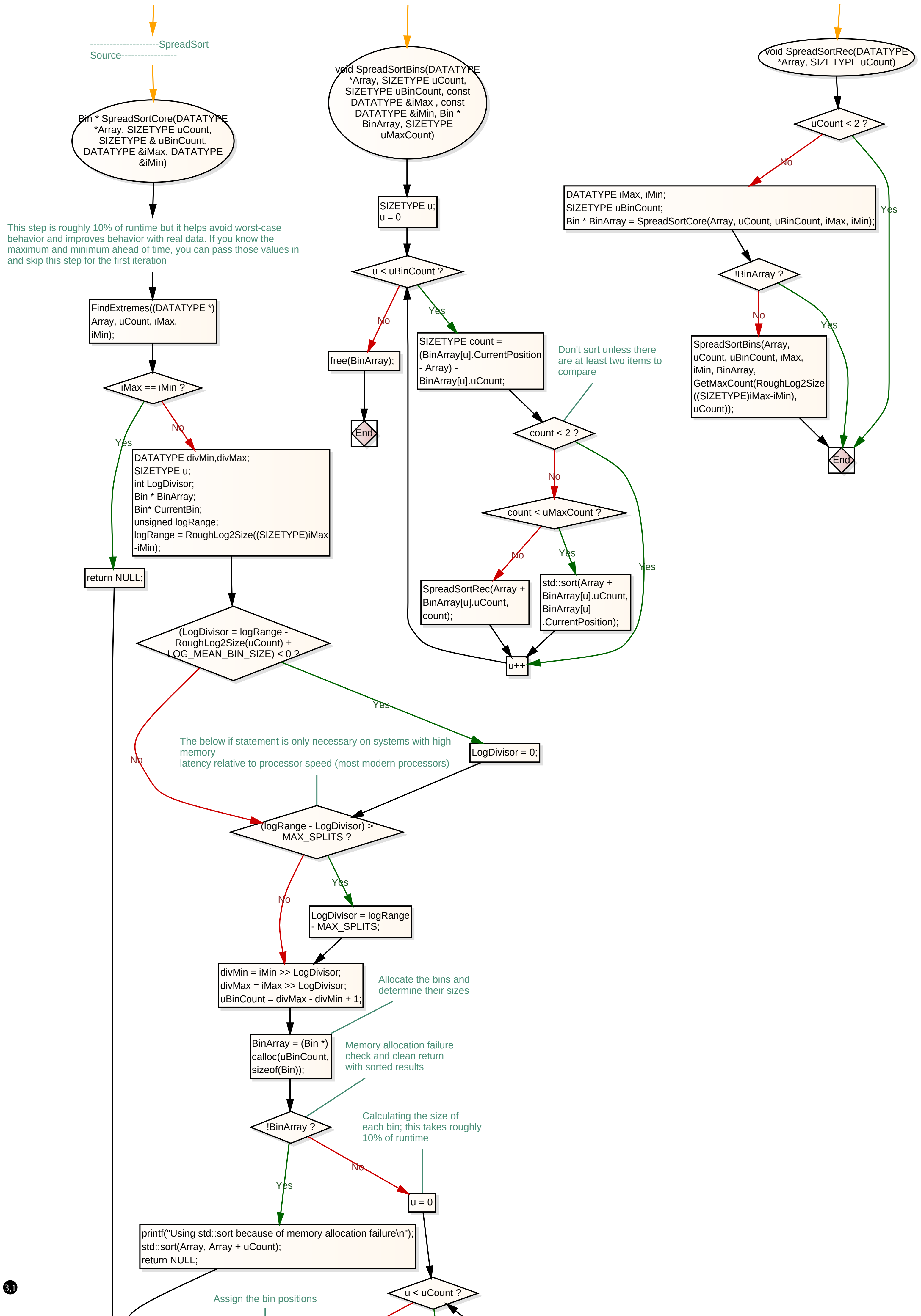
Spreadsort in C
Source: http://en.wikipedia.org/wiki/Spreadsort
This code is licensed under the Creative Commons Attribution-ShareAlike License.
It is from the Wikipedia article "Spreadsort" dated 2009-06-01.
Spreadsort is a sorting algorithm invented by Steven J. Ross in 2002.
It combines concepts from distribution-based sorts, such as radix sort
and bucket sort, with partitioning concepts from comparison sorts such
as quicksort and mergesort. In experimental results it was shown to be
highly efficient, often outperforming traditional algorithms such as
quicksort, particularly on distributions exhibiting structure.




unsigned RoughLog2(DATATYPE input)
unsigned char cResult = 0;
The && is necessary on some compilers to avoid infinite loops; it doesn't significantly impair performance
input >= 0 ?
Yes
No
(input >> cResult) && (cResult < DATA_SIZE) ?
((input >> cResult) < -1) && (cResult < DATA_SIZE) ?
Yes
No
No
Yes
cResult++;
return cResult;
cResult++;
End


SIZETYPE GetMaxCount(unsigned logRange, unsigned uCount)
unsigned logSize = RoughLog2Size(uCount);
unsigned uRelativeWidth = (LOG_CONST * logRange)/((logSize > MAX_SPLITS) ? MAX_SPLITS : logSize);
Don't try to bitshift more than the size of an element
DATA_SIZE <= uRelativeWidth ?
Yes
uRelativeWidth = DATA_SIZE - 1;
No
return 1 << ((uRelativeWidth < (LOG_MEAN_BIN_SIZE + LOG_MIN_SPLIT_COUNT)) ? (LOG_MEAN_BIN_SIZE + LOG_MIN_SPLIT_COUNT) : uRelativeWidth);
End
void FindExtremes(DATATYPE *Array, SIZETYPE uCount, DATATYPE & piMax, DATATYPE & piMin)
SIZETYPE u = 1;
piMin = piMax = Array[0];
u < uCount ?
Yes
No
End
Array[u] > piMax ?
Yes
No
Array[u] < piMin ?
piMax=Array[u];
Yes
No
piMin= Array[u];
++u


---------------------SpreadSort Source-----------------
Yes
No
iMax == iMin ?
return NULL;
(LogDivisor = logRange - RoughLog2Size(uCount) + LOG_MEAN_BIN_SIZE) < 0 ?
(logRange - LogDivisor) > MAX_SPLITS ?
!BinArray ?
u < uBinCount ?
count < 2 ?
count < uMaxCount ?
u++
uCount < 2 ?
End
u < uCount ?
3,1

```
Bin * SpreadSortCore(DATATYPE *Array, SIZETYPE uCount, SIZETYPE & uBinCount, DATATYPE &iMax, DATATYPE &iMin)
```

This step is roughly 10% of runtime but it helps avoid worst-case behavior and improves behavior with real data. If you know the maximum and minimum ahead of time, you can pass those values in and skip this step for the first iteration

```
FindExtremes((DATATYPE *) Array, uCount, iMax, iMin);
```

```
DATATYPE divMin,divMax;
SIZETYPE u;
int LogDivisor;
Bin * BinArray;
Bin* CurrentBin;
unsigned logRange;
logRange = RoughLog2Size((SIZETYPE)iMax -iMin);
```

```
LogDivisor = 0;
```

The below if statement is only necessary on systems with high memory latency relative to processor speed (most modern processors)

```
LogDivisor = logRange - MAX_SPLITS;
```

```
divMin = iMin >> LogDivisor;
divMax = iMax >> LogDivisor;
uBinCount = divMax - divMin + 1;
```

Allocate the bins and determine their sizes

```
BinArray = (Bin *) calloc(uBinCount, sizeof(Bin));
```

Memory allocation failure check and clean return with sorted results

```
printf("Using std::sort because of memory allocation failure\n");
std::sort(Array, Array + uCount);
return NULL;
```

Calculating the size of each bin; this takes roughly 10% of runtime

```
u = 0
```

Assign the bin positions

```
void SpreadSortBins(DATATYPE *Array, SIZETYPE uCount, SIZETYPE uBinCount, const DATATYPE &iMax , const DATATYPE &iMin, Bin * BinArray, SIZETYPE uMaxCount)
```

```
SIZETYPE u;
u = 0
```

```
free(BinArray);
```

```
SIZETYPE count = (BinArray[u].CurrentPosition - Array) - BinArray[u].uCount;
```

Don't sort unless there are at least two items to compare

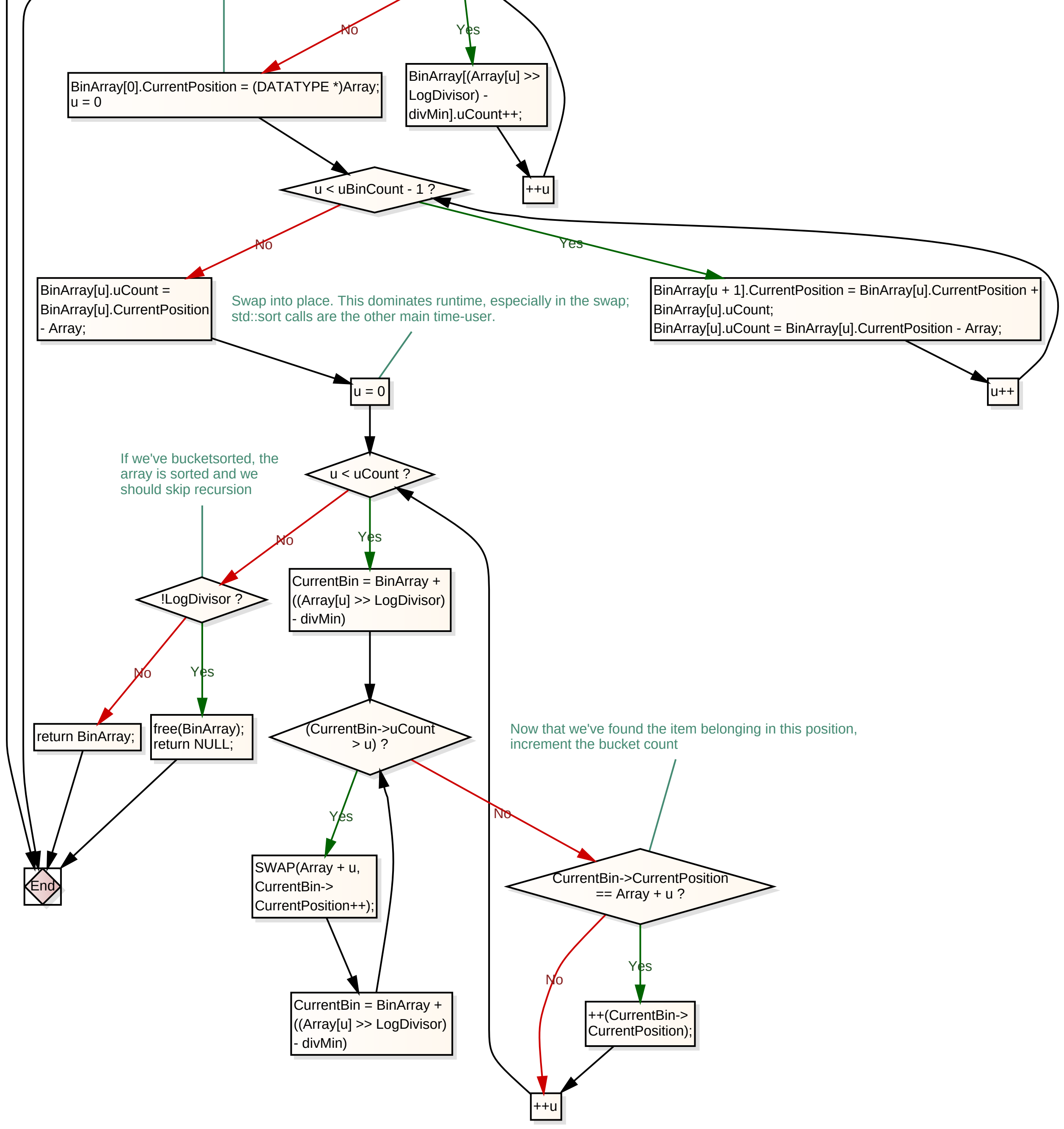
```
SpreadSortRec(Array + BinArray[u].uCount, count);
```

```
std::sort(Array + BinArray[u].uCount, BinArray[u] .CurrentPosition);
```

```
void SpreadSortRec(DATATYPE *Array, SIZETYPE uCount)
```

```
DATATYPE iMax, iMin;
SIZETYPE uBinCount;
Bin * BinArray = SpreadSortCore(Array, uCount, uBinCount, iMax, iMin);
```

```
SpreadSortBins(Array, uCount, uBinCount, iMax, iMin, BinArray, GetMaxCount(RoughLog2Size ((SIZETYPE)iMax-iMin), uCount));
```



No
Yes
BinArray[0].CurrentPosition = (DATATYPE *)Array;
u = 0
BinArray[(Array[u] >> LogDivisor) - divMin].uCount++;
++u
u < uBinCount - 1 ?
No
Yes
BinArray[u].uCount = BinArray[u].CurrentPosition - Array;
Swap into place. This dominates runtime, especially in the swap; std::sort calls are the other main time-user.
BinArray[u + 1].CurrentPosition = BinArray[u].CurrentPosition + BinArray[u].uCount;
BinArray[u].uCount = BinArray[u].CurrentPosition - Array;
u = 0
u++
If we've bucketsorted, the array is sorted and we should skip recursion
u < uCount ?
No
Yes
!LogDivisor ?
CurrentBin = BinArray + ((Array[u] >> LogDivisor) - divMin)
No
Yes
return BinArray;
free(BinArray);
return NULL;
(CurrentBin->uCount > u) ?
Now that we've found the item belonging in this position, increment the bucket count
Yes
No
End
SWAP(Array + u, CurrentBin->CurrentPosition++);
CurrentBin->CurrentPosition == Array + u ?
Yes
No
CurrentBin = BinArray + ((Array[u] >> LogDivisor) - divMin)
++(CurrentBin->CurrentPosition);
++u