GamaSort, an improved Radix Sort Exchange Algorithm

This is a great algorithm for small arrays, arrays with small numbers or with large numbers but within a small range of values such as unsigned integers (it will only work with positive integers).

For an array of integers within a range 0…k, or n…n+k the order of magnitude is $O(n\log_2 k)$. Actually, the use of a cutoff point and an insertion sort to finish the process makes it much faster but it also makes it vulnerable to "worst cases". The worst case of all would be an array with each element having bits 5 and above complementary to the ones of the next element and bits 0 to 4 being inversely sorted in a series that would repeat itself in the array. The probabilities of finding an array like this, or with similar structure are very low.

This algorithm is internal, recursive and not stable. It is not appropriate for arrays containing negative numbers or a very wide range of numbers.

In the real world the numbers obtained from sampling, scientific measurements or even indexes might be limited to small ranges. When sampling, it is common to have only natural numbers. This algorithm could be handy for such situations.

I was surprised with the performance of this algorithm; it is very fast for small arrays and great for a small k.

The idea of developing such algorithm happened by accident. When testing several sorting algorithms for a class (Advanced Algorithms), I had only a very simple description of the Radix Sort Exchange Algorithm and I coded it from there. When coding it some "shortcuts" were made to simplify it. That is when the idea of finding the lowest and highest bits to be scanned and two pivots to search for bits.

I would recommend checking out: "A library of internal sorting routines" by *Ariel Faigon* (http://www.yendor.com/programming/sort/)

The "divide and conquer" method used in this algorithm has nothing to do with QuickSort! C.A.R. Hoare developed QuickSort and it is considered as the most efficient sorting algorithm. QuickSort works with a pivot point which is an element taken from the array. Then it arranges the data so that all elements less than the pivot will be at its left and the others will be at its right.

The array is partitioned in two blocks and each block will recursively go through the same process of partitioning until 2 or fewer items are in one block and can be easily sorted. We can divide n items in half $\log_2 n$ times. This makes quicksort a O($n\log_2 n$) algorithm (for both best and average cases). The worst case for QuickSort is n^2 and it happens when the partitioning process divides a block into one single element in one side and the rest of the block in the other side. It happens for already sorted arrays.

Radix Exchange Sort should not be confused with Radix Sort which works like Bucket Sort, simply placing each element on it's own "bucket" or array position.
It is an O(n) algorithm as it takes only one straight loop:

```
for (int c=0; c< NumberOfElements; c++)
        DestinationArray[c]=SourceArray[c];
```

Radix Exchange Sort checks every bit from every element in the unsorted array starting by the most significant and going all the way down to the least significant. It compares those bits and moves all 0's to the left and all 1's to the right. Then it partitions the array and recursively go through the same process of partitioning using the next lower bit each time until the least significant bit. As each block is divided by 2 this makes Radix Exchange Sort a O($n\log_2 n$) algorithm, with a worst, average and best case of O($n\log_2 n$). To make this algorithm O($n\log_2 n$) only requires two minor changes in the code:

1) Commenting Call InSort(t(), 1, up)

2) Commenting (up-lo>CUTOFF)AND

Another change that can improve performance for small arrays of when the numbers are not within a certain range is to ignore GamaSortStarter and call GamaSort directly followed by Insertion Sort

```
GamaSort(array, 0, up, 0,7);//
InSort(array, up);//if (up-lo>CUTOFF)&& was removed then this line should be removed as well
```

My Improvement consists on 4 techniques:

1- Determining the maximum and minimum radices and sorting within this range to avoid extra scanning on empty areas. That is why O($n\log_2 k$) can be obtained.
2- Checking for all 1's or 0's to skip swapping and go to the next radix.
3- Using 2 pivots that will point to the 0's in the right and the 1's in the left, starting from each side and moving towards the center. This way, the number of comparisons and swapping are reduced.
4- Leaving small sets unsorted and using Insertion Sort to sort the whole array later. Insertion Sort is approximately O(n) for nearly sorted arrays and so it is very effective.

Here is the source of GamaSort:

```
#define SWAP(x, y) temp = (x); (x) = (y); (y) = temp//Swaps the contents of 2 variables
#define GT(x, y) ((x) > (y))//Greater Than
#define CUTOFF 15 //above this number there will be 2 recursive calls with 2 sub arrays
#define ArrayDimension 100000 //how many numbers to be tested
void  insort (int array[], int len)
{
        register int        i, j;
        register int        temp;

        for (i = 1; i < len; i++) {
                /* invariant:  array[0..i-1] is sorted */
                j = i;
                /* customization bug: SWAP is not used here */
                temp = array[j];
                while (j > 0 && GT(array[j-1], temp)) {
                        array[j] = array[j-1];
                        j--;
                }
                array[j] = temp;
        }
}

//
// Recursive version of GamaSort
//
static void GamaSort( int array[], int lo, int up, int lbit, int ubit )
//lo and up are the indexes of the sub-array to be ordered
//lbit is lowest bit to be scanned, if all elements of the array are >8, lbit=3
//ubit is the current bit being scanned
{
        register int        temp;
   register long       b, r, l;
        //b is the integer that is 2 to the power of the current bit being scanned
        //r is the index of the right pivot
        //l is the index of the left pivot
        if ((up-lo>CUTOFF)&&(ubit>=lbit)){//(up-lo>CUTOFF)&& can be removed if k log n is desired for all cases
                //keep going only if
                //a)the sub-array is greater than the cutoff point
                //b)the current bit is not smaller than the lowest bit
```

```
                        b=(1<<ubit);//b=2^ubit
                        r=up;l=lo;//2 pivots
                        while(((b & array[r])==b)&&(r>lo))
                                r--;//check 1's from the right
                        if (!((r==lo)&&((b & array[r])==b)))
                                while(((b & array[l])==0)&&(l<up))l++;//check 0's from the left
                        if(((r==lo)&&((b & array[r])==b))||((l==up)&&((b & array[l])==0)))
                                GamaSort(array, lo, up, lbit, ubit-1);//if it's all 0's or 1's, skip it
                        else{
                                //this is where the swapping occurs
                                while(l<r){
                                        if(((b & array[l])==b)&&((b & array[r])==0)){
                                                SWAP(array[r],array[l]);
                                                while(((b & array[r])==b)&&(r>lo))
                                                        r--;
                                        }
                                l++;
                        }
                        if (l>=r)
                                r++;
                        //recursivelly sort the 2 blocks corresponding to the sorted
                        //sub-arrays but now for a lower radix
                        if(r-1>lo)
                                GamaSort(array, lo, r-1, lbit, ubit-1 );//left
                        if (up>r)
                                GamaSort(array, r, up , lbit, ubit-1 );//right
        }
        }
}

//
// Main version of gamasort (call this one)
//
void GamaSortStarter( int  array[], int up )
{
        register int  i,ubit=1,counter,n=0,lbit=0,l=1;
        for (counter=0;counter<=up;counter++)
                n|=array[counter];
        if((n & 1)==0){
                i=1;
                while((n & (1<<i))==0)i++;
                lbit=i;
        }
        while((1<<ubit)<=n)
                ubit++;
        ubit--;
        GamaSort(array, 0, up, lbit,ubit);
        insort(array, up);//if (up-lo>CUTOFF)&& was removed then this line should be removed as well
                                                        //if k log n is desired for all cases
}
```

Acknowledgments:
To Professor Aaron Gordon-thank you for your help and for your challenging (and rewarding) classes.
To Mr. Ariel Faigon-thank you for the encouragement to improve my work and for testing and beautifying my code.

References:

"Algorithms", by Thomas C. Cormen, Charles E. Leiserson and Ronald L. Rivest
McGrawHill

"A library of internal sorting routines" by Ariel Faigon
(http://www.yendor.com/programming/sort/)

"Algorithm Archive", by Scott Gasch
(http://perl.guru.org/alg/node1.html)

"Citadel of the Sourcerer", by Nils Pipenbrinck
(http://www.cubic.org/~submissive/sourcerer/radix.htm)