The call produces 3 words of stack per recursive call and is able to take advantage of its knowledge of its own behavior. A more efficient implementation would sort small ranges by a more efficient method. If an implementation obeying standard calling conventions were needed, a simple wrapper could be written for the initial call to the above function that saves the appropriate registers.

AutoIt v3

This is a straightforward implementation based on the AppleScript example. It is certainly possible to come up with a more efficient one, but it will probably not be as clear as this one:

```
Func sort( ByRef $array, $left, $right )
  $i = $left
  $j = $right
  $v = $array[Round( ( $left + $right ) / 2 ,0)]
  While ( $j > $i )
      While ($array[$i] < $v )
          $i = $i + 1
      WEnd
      While ( $array[$j] > $v )
          $j = $j - 1
      WEnd
      If ( NOT ($i > $j) ) then
          swap($array[$i], $array[$j])
          $i = $i + 1
          $j = $j - 1
      EndIf
  WEnd
  if ( $left  < $j ) then sort( $array, $left, $j  )
  if ( $right > $i ) then sort( $array, $i, $right )
EndFunc
```

C

The implementation in the core implementations section is limited to arrays of integers. The following implementation works with any data type, given its size and a function that compares it. This is similar to what ISO/POSIX compliant C standard libraries provide:

```
#include <stdlib.h>
#include <stdio.h>

static void swap(void *x, void *y, size_t l) {
   char *a = x, *b = y, c;
   while(l--) {
      c = *a;
      *a++ = *b;
      *b++ = c;
   }
}

static void sort(char *array, size_t size, int (*cmp)(void*,void*), int begin, int end) {
   if (end > begin) {
      void *pivot = array + begin;
      int l = begin + size;
      int r = end;
      while(l < r) {
         if (cmp(array+l,pivot) <= 0) {
            l += size;
         } else if ( cmp(array+r, pivot) > 0 )  {
            r -= size;
         } else if ( l < r ) {
            swap(array+l, array+r, size);
         }
      }
      l -= size;
      swap(array+begin, array+l, size);
      sort(array, size, cmp, begin, l);
      sort(array, size, cmp, r, end);
   }
}

void qsort(void *array, size_t nitems, size_t size, int (*cmp)(void*,void*)) {
   sort(array, size, cmp, 0, nitems*size);
}

typedef int type;

int type_cmp(void *a, void *b){ return (*(type*)a)-(*(type*)b); }

int main(void){ /* simple test case for type=int */
  int num_list[]={5,4,3,2,1};
  int len=sizeof(num_list)/sizeof(type);
  char *sep="";
  int i;
  qsort(num_list,len,sizeof(type),type_cmp);
  printf("sorted_num_list={");
  for(i=0; i<len; i++){
    printf("%s%d",sep,num_list[i]);
    sep=", ";
  }
  printf("};\n");
  return 0;
}
```

Result:

```
sorted_num_list={1, 2, 3, 4, 5};
```

Here's yet another version with various other improvements:

```
/*****   macros create functional code   *****/
#define pivot_index() (begin+(end-begin)/2)
#define swap(a,b,t) ((t)=(a),(a)=(b),(b)=(t))

void sort(int array[], int begin, int end) {
   /*** Use of static here will reduce memory footprint, but will make it thread-unsafe ***/
   static int pivot;
   static int t;     /* temporary variable for swap */
   if (end > begin) {
      int l = begin + 1;
      int r = end;
      swap(array[begin], array[pivot_index()], t); /*** choose arbitrary pivot ***/
      pivot = array[begin];
      while(l < r) {
         if (array[l] <= pivot) {
            l++;
         } else {
            while(l < --r && array[r] >= pivot) /*** skip superfluous swaps ***/
               ;
            swap(array[l], array[r], t);
         }
      }
      l--;
      swap(array[begin], array[l], t);
      sort(array, begin, l);
      sort(array, r, end);
   }
}

#undef swap
#undef pivot_index
```

An alternate simple C quicksort. The first C implementation above does not sort the list properly if the initial input is a reverse sorted list, or any time in which the pivot turns out be the largest element in the list. Here is another sample quick sort implementation that does address these issues. Note that the swaps are done inline in this implementation. They may be replaced with a swap function as in the above examples.

```
void quick(int array[], int start, int end){
    if(start < end){
        int l=start+1, r=end, p = array[start];
        while(l<r){
            if(array[l] <= p)
                l++;
            else if(array[r] >= p)
                r--;
            else
                swap(array[l],array[r]);
        }
        if(array[l] < p){
            swap(array[l],array[start]);
            l--;
        }
        else{
            l--;
            swap(array[l],array[start]);
        }
        quick(array, start, l);
        quick(array, r, end);
    }
}
```

This sorts an array of integers using quicksort with in-place partition.

```
void quicksort(int x[], int first, int last) {
    int pivIndex = 0;
    if(first < last) {
        pivIndex = partition(x,first, last);
        quicksort(x,first,(pivIndex-1));
        quicksort(x,(pivIndex+1),last);
    }
}

int partition(int y[], int f, int l) {
    int up,down,temp;
    int piv = y[f];
    up = f;
    down = l;
    goto partLS;
    do {
        temp = y[up];
        y[up] = y[down];
        y[down] = temp;
    partLS:
        while (y[up] <= piv && up < l) {
            up++;
        }
        while (y[down] > piv  && down > f ) {
            down--;
        }
    } while (down > up);
    y[f] = y[down];
    y[down] = piv;
    return down;
}
```

The following sample of C code can be compiled to sort a vector of strings (defined as char *list[ ]), integers, doubles, etc. This piece of code implements a mixed iterative-recursive strategy that avoids out of stack risks even in worst case. It runs faster than the standard C lib function qsort(), especially when used with partially sorted arrays (compiled with free Borland bcc32 and tested with 1 million strings vector).

```
/********** QuickSort(): sorts the vector 'list[]' **********/

/**** Compile QuickSort for strings ****/
#define QS_TYPE char*
#define QS_COMPARE(a,b) (strcmp((a),(b)))

/**** Compile QuickSort for integers ****/
//#define QS_TYPE int
//#define QS_COMPARE(a,b) ((a)-(b))

/**** Compile QuickSort for doubles, sort list in inverted order ****/
//#define QS_TYPE double
//#define QS_COMPARE(a,b) ((b)-(a))

void QuickSort(QS_TYPE list[], int beg, int end)
{
    QS_TYPE piv; QS_TYPE tmp;

    int  l,r,p;

    while (beg<end)    // This while loop will avoid the second recursive call
    {
        l = beg; p = beg + (end-beg)/2; r = end;

        piv = list[p];

        while (1)
        {
            while ( (l<=r) && ( QS_COMPARE(list[l],piv) <= 0 ) ) l++;
            while ( (l<=r) && ( QS_COMPARE(list[r],piv)  > 0 ) ) r--;

            if (l>r) break;

            tmp=list[l]; list[l]=list[r]; list[r]=tmp;

            if (p==r) p=l;

            l++; r--;
        }

        list[p]=list[r]; list[r]=piv;
        r--;

        // Recursion on the shorter side & loop (with new indexes) on the longer
        if ((r-beg)<(end-l))
        {
            QuickSort(list, beg, r);
            beg=l;
        }
        else
        {
            QuickSort(list, l, end);
            end=r;
        }
    }
}
```

Iterative Quicksort

Quicksort could also be implemented iteratively with the help of a little stack. Here a simple version with random selection of the pivot element:

```
typedef long type;                                       /* array type */
#define MAX 64            /* stack size for max 2^(64/2) array elements  */

void quicksort_iterative(type array[], unsigned len) {
   unsigned left = 0, stack[MAX], pos = 0, seed = rand();
   for ( ; ; ) {                                           /* outer loop */
      for (; left+1 < len; len++) {                 /* sort left to len-1 */
         if (pos == MAX) len = stack[pos = 0];  /* stack overflow, reset */
         type pivot = array[left+seed%(len-left)];  /* pick random pivot */
         seed = seed*69069+1;                /* next pseudorandom number */
         stack[pos++] = len;                    /* sort right part later */
         for (unsigned right = left-1; ; ) { /* inner loop: partitioning */
            while (array[++right] < pivot);  /* look for greater element */
            while (pivot < array[--len]);    /* look for smaller element */
            if (right >= len) break;           /* partition point found? */
            type temp = array[right];
            array[right] = array[len];                  /* the only swap */
            array[len] = temp;
         }                            /* partitioned, continue left part */
      }
      if (pos == 0) break;                               /* stack empty? */
      left = len;                             /* left to right is sorted */
      len = stack[--pos];                      /* get next range to sort */
   }
}
```

The pseudorandom selection of the pivot element ensures efficient sorting in O(n log n) under all input conditions (increasing, decreasing order, equal elements). The size of the needed stack is smaller than $2 \cdot \log_2(n)$ entries (about 99.9% probability). If a limited stack overflows the sorting simply restarts.

C++

This is a generic, STL-based version of quicksort.

Note that this implementation uses last iterator content, and is not suitable for a std::[whatever]sort replacement as is.

```
#include <functional>
#include <algorithm>
#include <iterator>

template< typename BidirectionalIterator, typename Compare >
void quick_sort( BidirectionalIterator first, BidirectionalIterator last, Compare cmp ) {
    if( first != last ) {
        BidirectionalIterator left  = first;
        BidirectionalIterator right = last;
        BidirectionalIterator pivot = left++;

        while( left != right ) {
            if( cmp( *left, *pivot ) ) {
                ++left;
            } else {
                while( (left != right) && cmp( *pivot, *right ) )
                    --right;
                std::iter_swap( left, right );
            }
        }

        --left;
        std::iter_swap( pivot, left );

        quick_sort( first, left, cmp );
        quick_sort( right, last, cmp );
    }
}

template< typename BidirectionalIterator >
    inline void quick_sort( BidirectionalIterator first, BidirectionalIterator last ) {
        quick_sort( first, last,
                std::less_equal< typename std::iterator_traits< BidirectionalIterator >::value_type >()
                );
    }
```

Here's a shorter version than the one in the core implementations section which takes advantage of the standard library's partition() function:

```
#include <algorithm>
#include <iterator>
#include <functional>

using namespace std;

template <typename T>
void sort(T begin, T end) {
    if (begin != end) {
        T middle = partition (begin, end, bind2nd(
                    less<typename iterator_traits<T>::value_type>(), *begin));
        sort (begin, middle);
//        sort (max(begin + 1, middle), end);
        T new_middle = begin;
        sort (++new_middle, end);
    }
}
```

C#

The followings C# implementations uses the functional aspect of c#

```
public IEnumerable<T> Quicksort<T>(List<T> v, IComparer<T> comparer)
{
    if (v.Count < 2)
        return v;

    T pivot = v[v.Count / 2];

    return Quicksort(v.Where(x => comparer.Compare(x, pivot) < 0), comparer)
        .Concat(new T[] { pivot })
        .Concat(Quicksort(v.Where(x => comparer.Compare(x, pivot) > 0), comparer));
}
```

Faster cause uses partition

```
public IEnumerable<T> Quicksort(IEnumerable<T> v, Comparer<T> compare)
{
    if (!v.Any())
        return Enumerable.Empty<T>();

    T pivot = v.First();

    // partition
    Stack<T> lowers = new Stack<T>(), greaters = new Stack<T>();

    foreach (T item in v.Skip(1)) // skip the pivot
        (compare(item, pivot) < 0 ? lowers : greaters).Push(item);

    return Quicksort(lowers, compare)
        .Concat(new T[] { pivot })
        .Concat(Quicksort(greaters, compare));
}
```

The following example uses linq to filter the list

```
private void Quicksort<T>(T[] v, int left, int right, IComparer<T> comparer)
{
```