Implementing a Generic Quick Sort With Various Optimizations

I decided for educational purpose to make a generic sort like the one in the standard library as an exercise for my self in learning more about how to do generic programming in C and so I would really appreciate it if you guys could tell me how I could have done a better job and what other various improvements I could add to make my code better. I have quite a bit of functions to go through so I'm going to explain what each function does as much as I need to.

Main

```c
int main()
{
    /*
    Throughout this source code, if you see an unsigned int, more than likely
    I'm using an unsigned int because I'm comparing to a variable of type size_t
    */
    unsigned int i;
    int a[] = {200, 1, 99, 23, 56, 207, 369, 136, 147, 21, 4, 75, 36, 12};

    size_t numberOfElements = sizeof(a)/sizeof(a[0]);

    /*
     No point in performing quick sort if it does not have more than 1 elements and if
```

```
     the array is sorted.
    */
    if(numberOfElements > 1 && !isSorted(a, numberOfElements, sizeof(int), cmp))
    {
        quickSort_h(a, numberOfElements, sizeof(int), cmp);
    }

    for(i = 0; i < numberOfElements; i++)
    {
        printf("%d ", a[i]);
    }

    return 0;
}
```

Quick Sort Helper

```
void quickSort_h(void *base, size_t nitems, size_t memSize, int (*cmp)(const void *, const
void *))
{
    quickSort(base, 0, nitems - 1, memSize, cmp);
    insertionSort(base, nitems, memSize, cmp);
}
```

Quick Sort

```
/*
  Here we are doing a tail recurse optimization and using our threshold value(20) to know
  when to move to Insertion Sort.
*/
void quickSort(void *base, int first, int last, size_t memSize, int (*cmp)(const void *,
const void *))
{
    while(last - first > THRESHOLD)
    {
        int pivot;

        pivot = qSortPartition(base, first, last, memSize, cmp);

        /*
          Here we are checking which is easier to recurse by checking if the subarray
          on the left is shorter than the one on the right and vice versa.
        */
        if(pivot - first < last - pivot)
        {
           quickSort(base, first, pivot - 1, memSize, cmp);
           first = pivot + 1;
        }
        else
        {
          quickSort(base, pivot + 1, last, memSize, cmp);
```

```
            last = pivot - 1;
        }
    }
}
```

qSortPartition

```
/*
 Here we find the median of the first, middle and last element and we will use
 that as are starting pivot.
*/
int qSortPartition(void *base, int first, int last, size_t memSize, int (*cmp)(const void
*, const void *))
{
    char *carray = (char *)base;

    int pivot, mid = first + (last - first) / 2;

    // Find the larger of the two
    pivot = cmp(&carray[first * memSize], &carray[mid * memSize]) > 0 ? first : mid;

    // Find the smallest one.
    if(cmp(&carray[pivot * memSize], &carray[last * memSize]) > 0)
    {
        pivot = last;
    }

    // Put the pivot at the front of the list.
    byteSwap(&carray[pivot * memSize], &carray[first * memSize], memSize);
    pivot = first;

    while(first < last)
    {
        /*
           if the value we have is less than the element at the end,  move the pivot up
           by 1, else move first.
        */

        if(cmp(&carray[first * memSize], &carray[last * memSize]) <= 0)
        {
            byteSwap(&carray[pivot * memSize], &carray[first * memSize], memSize);
            pivot++;
        }

        first++;
    }

    /* finally swap the pivot with the last element. At this point, all elements to the
       left of the pivot are less than it and vice versa.
    */
    byteSwap(&carray[pivot * memSize], &carray[last * memSize], memSize);
```

```
    return pivot;
}
```

Insertion Sort

```
void insertionSort(void *base, size_t nitems, size_t memSize, int (*cmp)(const void *,
const void *))
{
    char *carray = (char *)base;
    unsigned int i;
    int j;

    for(i = 0; i < nitems; i++)
    {
        j = i;
        while(j > 0 && cmp(&carray[j * memSize], &carray[(j - 1) * memSize]) < 0)
        {
            byteSwap(&carray[j * memSize], &carray[(j - 1) * memSize], memSize);
            j--;
        }
    }
}
```

isSorted

```
int isSorted(void *base, size_t nitems, size_t memSize, int (*cmp)(const void *, const
void *))
{
    char *carray = (char *)base;
    unsigned int i;

    for(i = 0; i < nitems - 1; i++)
    {
        // Simply check if the current element is greater than the next element.
        if(cmp(&carray[i * memSize], &carray[(i + 1) * memSize]) > 0)
        {
            return 0;
        }
    }

    return 1;
}
```

compare

```
int cmp(const void *a, const void *b)
{
   const int *A = a, *B = b;
   // A > B = 1, A < B = -1, (A == B) = 0
   return (*A > *B) - (*A < *B);
}
```

byteSwap

```
/*
  The main way that this function is swapping is by swapping out the bytes in each
  iteration.
*/
void byteSwap(void *a, void *b, size_t memSize)
{
    char tmp;
    char *aa = a, *bb = b;

    do
    {
        tmp = *aa;
        *aa++ = *bb;
        *bb++ = tmp;
    }
    while(--memSize > 0);
}
```

c generics quick-sort

edited Sep 6 '16 at 18:28

ferada
8,531 13 53

asked Sep 6 '16 at 14:09

Luis Averhoff
83 1 7

Are you using any specific C standard? – Johnbot Sep 6 '16 at 14:13

@Johnbot I've set my compiler to follow the C89 Standard with the -pedantic warning on which is why you see a lot of the variable starting at the top. – Luis Averhoff Sep 6 '16 at 14:19

Your code to choose a median of three for pivot is incorrect. – CiaPan Sep 14 '16 at 8:20

@CiaPan Can you explain why it is incorrect? – Luis Averhoff Sep 14 '16 at 21:13

@LuisAverhoff Your algorithm in short is `pivot = min(max(first,mid),last);` and it will return the `last` value not only if it's a median, but also when it is the smallest one among the three. – CiaPan Sep 15 '16 at 7:36

1 Answer

Don't guess at compatibility

```
/*
Throughout this source code, if you see an unsigned int, more than likely
I'm using an unsigned int because I'm comparing to a variable of type size_t
*/
unsigned int i;
```

Why not just use `size_t` ?

```
size_t i;
```

As stands, this has the worst of both worlds. You are using `unsigned int` because it usually matches `size_t` . But it doesn't always match `size_t` . And it doesn't allow you to set `i` to negative values. So you may have to pay the conversion penalty if it doesn't match. And you have to accept the limitations of an unsigned type regardless. If you use `size_t` instead, you have to accept the limitations but at least you are guaranteed not to have to do a conversion.

Know your bounds

```
for(i = 0; i < nitems; i++)
{
    j = i;
    while(j > 0 && cmp(&carray[j * memSize], &carray[(j - 1) * memSize]) < 0)
```

Consider

```
for (i = 1; i < nitems; i++)
{
    j = i;
    while(j > 0 && cmp(&carray[j * memSize], &carray[(j - 1) * memSize]) < 0)
```

That saves one iteration of the outer loop without changing behavior. This is because the inner loop won't run when `i` and `j` are 0.

Don't subtract and multiply when you can just do one

```
char *carray = (char *)base;
unsigned int i;
int j;

for(i = 0; i < nitems; i++)
{
    j = i;
    while(j > 0 && cmp(&carray[j * memSize], &carray[(j - 1) * memSize]) < 0)
```

```
        {
            byteSwap(&carray[j * memSize], &carray[(j - 1) * memSize], memSize);
            j--;
        }
    }
```

Consider

```
char *start = base;
char *end = start + nitems * memSize;
char *top;

for (top = start + memSize; top < end; top += memSize)
{
    char *previous = top;
    char *current = previous - memSize;

    while (previous > start && cmp(previous, current) > 0)
    {
        byteSwap(previous, current, memSize);

        previous = current;
        current -= memSize;
    }
}
```

This version only does one explicit multiplication in the entire thing. Rather than working with indexes that need to be converted into pointers, this manipulates the pointers directly.

Note that `&start[nitems * memSize]` is equivalent to `start + nitems * memSize`. That will be true even for arrays of things that are larger than one byte per element. C compilers are smart enough that when they add an integer to a pointer, they automatically multiply by the width of the element. You wouldn't need `memSize` at all if casting to `void *` hadn't discarded the type information.

Also note that `cmp` and `byteSwap` are guaranteed to work with the same values now. The original version did the same operations twice each. Doing the same thing in parallel is risky. It allows for edits that only change something in one place that needs to be changed in two.

Another side effect of this is that we only calculate one value per iteration. The original version did at least two and possibly four absent compiler optimizations.

Declarations

Even in older C versions, you aren't limited to declaring variables at the start of a function. You are limited to the start of a block.

Similarly,

```
char tmp;
char *aa = a, *bb = b;

do
{
    tmp = *aa;
```

could be

```
char *aa = a;
char *bb = b;

do
{
    char tmp = *aa;
```

Note that this also changes the declarations so that all assignments are alone. It's easy to miss a declaration or assignment if more than one is done per line. A rule limiting to one declaration per line with an assignment increases readability.

edited Sep 7 '16 at 13:27

answered Sep 6 '16 at 17:48

mdfst13

15.7k 4 17 53

Great advice all around. Though I will like to add that there was two small errors that you made in the modified while loop you showed me. You had current > start when it should have been current >= start because if you have this situation {5, 4} current is not greater than in this situation it is actually equal. The second error you made was that you had a greater than symbol instead of a less than symbol. Again refer to my above example again. Overall, I did enough reading your suggestions. – Luis Averhoff Sep 7 '16 at 4:22