Code Optimization

In this exercise, we're going to learn a valuable lesson. Making small, local changes in your code can help improve performance, but an optimizing compiler will generally do a better job of this than you can. As an intelligent software developer, your time is better spent thinking about refining the data structures and algorithms used in your program. An optimizing compiler won't do this for you, and it can have a much greater impact on performance than code optimization or the choice of a programming language.

On the course homepage you'll find files for this exercise. You can also download them with the following curl commands:

curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise23/bubble.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise23/merge.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise23/big_input.txt

You'll find a copy of the bubble sort program we looked at on day 23 of class (with the small bug in the sort already fixed). First, compile and run this program with the big_input.txt file as input. Time its execution on this input. For me, it took about 2 seconds to run on the common platform machines at remote.ecs.ncsu.edu:

```
gcc -Wall -std=c99 bubble.c -o bubble
eos$ time ./bubble < big_input.txt
```

Now, copy this source file to a file named **bubble2.c**. Try inlining the swap() function by hand. Replace the call to swap(), with a copy of the swap() function's body. You may have to modify it a little to get it to work in place of the call. As we saw in class, this function gets called about 100 million times sorting this input, so saving the function call overhead should make a difference in performance. Compile and run this version of the bubble sort program and see what you get. For me, this reduced the execution time by about 20 percent on the same common platform machine.

Now, copy bubble2.c to bubble3.c and let's make another change. The inner loop in the sortList() function is where we spend most of our computation time. We'll use strength reduction and code motion to speed up this loop a little bit more.

- First, let's reduce the overhead for array indexing. Change the loop control variable from an integer, **j**, to an int pointer, **p**. At the start of the loop, initialize **p** to point to the first element of the list, and, on each iteration, just increment **p** so it moves ahead to point to the next item. Then, when we want to access an item, we don't need to do array indexing with a variable like **j**, we can just use ***p** to access the current element and something like **p[1]** to access its successor.
  I'd say this optimization is an example of strength reduction since we're replacing array indexing like **p[j]** (basically an add operation and a dereference) with ***p** (just a dereference).
- Next, let's simplify the condition for the loop (we have to change this condition anyway, since we changed the loop control variable). Instead of computing **len - i - 1** over and over for every information, let's compute the address of **list[len - i - 1]** before we even start the loop and store that address in a new local variable named something like **end**. Then, the condition on our loop can be something simple, like **p < end**.
  I'd say this is an example of code motion because we're moving the repeated calculation of **len - i - 1** outside the loop.

Try compiling and timing the execution of this version of your program. When I tried this, it sped up my execution by another 5 or 6 percent compared to **bubble2.c**.

It's nice to get these performance improvements and to see how optimizations work, but I think we've been wasting our time. These small reorganizations in our code are a good job for the compiler (and they're a bad idea if they hurt the readability of our code). Let's go back to the original version of the program, bubble.c and see what the compiler can do. Compile with the -O2 flag and see what the execution time is like on the same input. When I tried this, I got an runtime that was considerably faster than my hand-optimized **bubble3.c**.

```
$ gcc -O2 -Wall -std=c99 bubble.c -o bubble
$ time ./bubble  < big_input.txt
```
Now, let's try spending our effort where it matters. With the other files, you'll find a source file, **merge.c**. It's just like **bubble.c**, but it has a partial implementation of a merge sort in place of the bubble sort. You get to complete this implementation. You remember merge sort. It takes an unsorted list, splits it into two smaller, unsorted lists, each about half as long, recursively mergesorts the two smaller lists, then merges them back into one, sorted list. I've already written about half of this function for you. You get to add the part where you split the list into two smaller lists and the part where you merge the sorted lists back together.

Once your merge sort is done, compile your program (without any optimization), and time its execution on the same, large input file. I don't want to give away the fun part of this exercise, but my solution was more than 20 times faster than my compiler-optimized bubble sort.

Submit all three of your modified source files, **bubble2.c**, **bubble3.c** and **merge.c** to the exercise_23 assignment on Moodle.