# ECE 264: Advanced C Programming

Programming Assignment 6 - Merge Sort, Due 7/15/19

In PA05 you wrote several routines to sort Student records that you read in from a file. Your job in that assignment was to write input/output methods to create an array of Student records, and write comparison functions to compare Students by different fields in their records. However, you used the C standard library's built-in `qsort` function to perform the actual sort. In this assignment, you will write your own sort function.

Note: This assignment builds on PA05, replacing `qsort` with your own sort, but otherwise leaving the rest of the assignment the same. If you were unable to complete PA05, or are concerned that your implementation of PA05 is incorrect, you have two options:

1. You can download pre-built object files for student.o and pa06.o. You can then link these files in your Makefile instead of compiling your own versions of student.c and pa06.c. If you take this option you will have to develop your code on the ecegrid machines.
2. On July 15th (after the late submission deadline for PA05 has passed), we will release student.c and pa06.c, which will be correct, working implementations of the code you needed to write for PA05.

## 1 Learning Goals

You will learn:

- Recursion principles
- "Divide-and-conquer" recursion
- Merge sort

## 2 Background

### 2.1 Recursion

In a superficial sense, recursion is what happens whenever a function `f` calls itself, as in the "standard" factorial example:

```
int factorial(int n) {
  if (n == 0) return 1;
  else return n * factorial(n - 1);
}
```

It is better to think of recursion as a *technique* for solving problems. Many problems can be thought of using the following pattern:

1. Break the problem up into "smaller" version(s) of the same problem.
2. Solve the smaller problem(s) by calling the same function (we call this the *inductive case*)
3. Use the solutions to the smaller problem(s) to solve the original problem.

This seems like a process that doesn't end: to solve a big problem, we break it up into smaller versions of the problem— but then we have to solve the smaller problem, which isn't any different! The key is that you can repeat this process, solving the smaller problems in the same way. At each step, you get smaller and smaller problems. Eventually, the problem is small enough that getting the answer is trivial. We call this the *base case*.

We can see this in the `factorial` example: rather than computing factorial of `n`, we realize that `n!` is just `n * (n-1)!` (Step 1: break the problem up into a smaller version of the same problem) – so we can call `factorial(n - 1)` (Step 2: solve the smaller problem by calling the same function). We can then multiply this by `n` to find `factorial(n)` (Step 3: use the solution of the smaller problem to solve the original problem). We also see that the base case is simple: we already know what 0! is, so there is no need to "break it up" into a smaller problem – we can just return 1.

(Note: you could also write factorial with a loop, and the loop version would probably be faster, so you might wonder why we need recursion. The code you will write in this assignment is a case where recursion is basically the only way to write it.)

One way to think about how to correctly write a recursive function is to think *inductively*: We can *assume* that the recursive function already works, but only if the function is called on a smaller problem than what we're solving. We can then write the recursive function assuming that it already works. The only thing we have to make sure we do is write correct base cases – we need to make sure that for the smallest versions of the function, we compute the correct answer. (This sounds circular, but it works for the same reason that inductive proofs work)

## 2.2 Divide-and-conquer Recursion

A very common pattern for recursive problems is *divide-and-conquer recursion*: to solve a problem on `n` pieces of data (e.g., an array of length n), we break the input up into two pieces, each with `n/2` pieces of data (e.g., two arrays, each with half the elements), call the recursive function on these smaller pieces, then write some code to combine the results from those two functions into the final answer. The base case for this style of function is what to do when you have only 1 element.

Consider a toy example where we want to sum up all the values in an input array with n elements. Here, if we divide the array in two and sum those two sub-arrays, we can add the results to get the sum of the whole array. The base case is that the sum of an array with just one element is the value of that element:

```
int sum(int * arr, int nels) {
  if (nels == 1) return arr[0];

  int sum1 = sum(arr, nels/2);
  int sum2 = sum(&arr[nels/2], (nels + 1)/2);

  return sum1 + sum2
}
```

(The `(nels + 1)/2` stuff is just a fancy way of dealing with arrays that have an odd number of elements, where `sum2` works over a slightly larger array than `sum1`. In integer division, `nels/2` is like computing `floor(n/2)`, and `(nels + 1)/2` is like computing `ceiling(n/2)`. More generally, to compute `ceiling(a/b)` you can do integer division: `(a + b - 1)/b.)`

## 2.3 Merge Sort

Your task in this programming assignment is to write a *merge sort*. Merge sort is an application of divide and conquer recursion to sort an array. The heart of merge sort is the `merge` operation, which combines two *already sorted* arrays to produce a new sorted array. To merge two sorted arrays, imagine you have two cursors, which start at the beginnings of the two arrays. Look at the two elements pointed to by the cursor: add whichever element is smaller to the output array, then move that cursor forward by 1 element. (If one of the cursors is already at the end of its array, the other cursor always "wins.")

This `merge` operation gives us a way of combining the solutions of two smaller problems to solve the larger problem of sorting an array:

1. Divide the array into two pieces
2. Sort the two pieces by recursively calling the same function
3. Use merge to merge the two resulting sorted pieces

So what should the base case be? How do we make sure we don't keep sorting smaller and smaller arrays? Note something simple: an array with only one element is already sorted!

## 3 What do you need to do?

Your job is to write a merge sort function, which you will call `msort`. To test `msort`, you will modify your PA05 submission to call `msort` instead of `qsort`. The signature of `msort` is:

```
void msort(Student * base, int nel, int (*compar)(const void *, const void *))
```

Where `compar` is a pointer to a comparison function (you can/should reuse your comparison functions from PA05).

As explained above, the heart of merge sort is a `merge` function, which you will also have to write. The signature of `merge` is:

```
Student * merge(Student * base1, int nel1, Student * base2, int nel2, int (*compar)(const void
*, const void *))
```

Where `merge` returns a newly allocated array of Students that is the result of merging the arrays `base1` and `base2`.

### 3.1 Files you need to modify

You will need to only modify and submit one file: `merge.c`, providing new definitions of `merge` and `msort`. To test your code, you will either need your own code from PA05 (don't forget to `#include "msort.h"` in your `student.c`) or you will need to use one of the options described at the beginning of this README.

### 3.2 Grading and partial credit

We will grade PA06 using the same inputs (and expected outputs) as PA05. Do not remove the `#ifndef` directives around `merge` and `msort`. We will test each of them separately for partial credit. (Note that `#ifndef` works the opposite way to `#ifdef` – if a particular flag is not defined, then the code in the `#ifndef` will be included.)

*We will deduct 20 points for all Git submission related errors starting from PA06.*