1 Sorting

In this lecture, instead of covering new subjects in C, we will stop and try to solidify our current knowledge by studying a well-known problem in programming. The problem is *sorting*. Sorting simply means to put the elements of a given set (which *may* contain duplicates) into ascending or descending order. The problem we will attack now is the problem of sorting a list of integers into ascending order. So, given the list:

```
45 67 32 14 2 88 141 13
```

We want the following result:

```
2 13 14 32 45 67 88 141
```

In order to achieve this result, we will write a function that has the following prototype:

```
void sort(int *list, int length);
```

Note that the function does not return any value. The first argument is a pointer to an `int`eger which should point to an array. If you remember our discussion of pointers, we can use this pointer to refer to any element of the array itself. One problem is that the pointer is not sufficient to know the length of the array in C. Therefore, the second argument should be used to pass the length of the array to the function. Here is an example of how this function might be called:

```
int main(void)
{
  int a[5] = {21, 3, 33, 143, 2};

  sort(a, 5);

}
```

So, we pass the array `a`, and its length 5 to our function. There is one pitfall here: You have been told that function parameters are always passed by value in C, and therefore they may not be modified by the function being called. However, any changes made to the array by the function here will be permanent. How does that happen? This can happen when you pass *pointers* to a function. The pointer itself can not be modified by the function, but any data that the pointer points to can actually be modified by the function.

2 The Algorithm

There are many known algorithms for sorting (or, comparison sorting to be precise). The one we will use here is one of the worst ways of doing this, called "selection sort". The algorithm goes like this:

```
1. Find the smallest member of the set.
2. Swap the smallest element with the first element.
3. If all elements are sorted, stop.
4. Ignoring all the sorted elements, go back to step 1.
```

This method is more easily demonstrated than written out, so do not be scared.

3 Writing the Code

3.1 Step One: Finding the Smallest Member of an Array

You can see that our simple algorithm contains a step which states "find the smallest member of the set". We will begin the solution of our problem by writing a function that does just that and finds the smallest member of a given set. The prototype of the function will be:

```
int index_of_smallest_element(int *set, int length);
```

The name of the function states that it returns the smallest element of the set. The value the function returns should just be the *index* of the smallest element in the set, and *not* the value of the element itself.

Just how do we do that? If you can not figure it out yourself, here is some information: First, note that you can not decide that an element is the smallest element without examining all of the elements in the array. So, it would seem that we need to loop over all the elements in the array.

We will also need a variable to hold the index of the smallest element, and we will eventually return the value of that variable. What will that variable's initial value be? I think it makes sense to set it to zero. In that way, we initially assume that the first element of the array is the smallest element.

In this case, now we have to check all the *other* elements in the array. So, it would make sense to start the loop variable from 1 rather than 0. Inside the loop, we need only check whether the element at the current index is smaller than the value of the element at our "assumed" index. If it is, then we just replace the "smallest element index" with the current index (i.e. the value of the loop variable).

Once the loop is done, we can return the index of the smallest element easily. Once you have written the function, make sure you test it to make sure it indeed returns the index of the smallest element in the array.

3.2 Step Two: Sorting: Not just yet...

Now we are ready to do our sorting. Or are we? In order to sort, we need to be able to ignore some starting elements in the array. For example, if we are looking for the third smallest, and we have already placed the two smallest elements at the beginning of the array, we want the smallest element *whose index is greater than or equal to two.* (Note that indices start at zero.)

Our function therefore has to know how many elements it needs to skip. To do that, we can add one more argument to our function making its prototype as follows:

```
int index_of_smallest_element(int *set, int length, int skip);
```

The argument `skip` here will tell our function how many elements to skip. In this case, just two changes are required in our function: First, start the "search" with element `skip` rather than zero, and start the loop at `skip` plus one rather than one.

3.3 Step Three: Finally, Sorting

Now, you should be ready to write the function that will do the actual sorting. You will be calling the function you wrote in the last section. At one point you will obviously need to swap two elements of the array. You have seen this before, but the way to do it is to use an auxilliary variable:

```
swap = a[i];
a[i] = a[j];
a[j] = swap;
```

You should be able to figure this out on your own.

4 Recursion

Recursion is a term you are bound to hear sooner or later in programming. It means roughly to "happen again" (I am no English major!). Recursion in programming is the name given to the technique where one or more functions repeatedly call each other or themselves. Such functions are called "recursive" functions.

Obviously, a function can not keep calling itself infinitely many times; you will run out of memory, time, or both. Normally, any recursive function has a well-defined stopping point. You can see recursive definitions in mathematics as well:

$$n! = n \cdot (n-1)!$$

Here, $n!$ is defined recursively, as n times $(n-1)!$. Does it have a definite stopping point? It sure does, and that point is defined by:

$$1! = 1$$

The obvious way to find the factorial of a number in C is as below:

```
int factorial (int n)
{
  int i;
  int fac = 1;

  for (i=2; i<=n; i++) {
    fac *= i;
  }

  return fac;
}
```

This will obviously work fine. But, can we possibly write it recursively? The answer is yes, of course. Here is a first attempt at writing it:

```
/* This will not work */

int factorial (int n)
{
  return n*factorial(n-1);
}
```

As the comment states, this will *not* work. Why not? Because it does not have a stopping point. The function will just keep calling itself forever. We need to supply the stopping point. Where is that? If the function is called with 1, then we no more call the function itself, but just return 1. Otherwise, recursion continues.

```
/* This will work */

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

}
```

At this point, you should type this program in, and *really* understand it.

5 Exercise: Another Example of Recursion

Another thing with a recursive definition is the Fibonacci series (that should sound familiar). The definition of the Fibonacci series is as below:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ F_1 &= 1 \\ F_2 &= 1 \end{aligned}$$

Now, you should be able to write a function that calculates the Fibonacci numbers using recursion. Note that the code will look much simpler.

6 Pros and Cons of Recursion

In fact, these two examples of recursion are things that you should never do in practice. The rule of thumb is that if something can be done using a loop, recursion is just wasteful, and slow.

Why is recursion wasteful? Since every time the computer calls a function, is has to remember where it was called from. So, it saves all local variables and execution information, then calls the function. When the function returns, local variables are restored and execution resumes. Each recursive call wastes some memory and causes a function-calling overhead.

On the other hand, recursion can be a very powerful tool under the right conditions. In fact, there are some situations where the only “proper” way to solve the problem is recursive. Hopefully, we’ll get to see such situations eventually.

7 How Fast Will It Run?

Perhaps the most important point in deciding how good a particular algorithm is how fast it actually runs. It is hard to find out how fast things will run by just looking at the algorithm, but it is possible to find out how it will change with "size of input" or "term number", as the case may be.

Let us give an example of how things go by looking at the recursive factorial function implementation. Assume that one function call and execution takes A seconds to execute. Until the factorial is calculated, the function will be called n times. Therefore, the running time of the factorial function can be said to be $A \cdot n$. Thus, the time it takes to calculate a factorial varies linearly with time.

How about the recursive Fibonacci implementation? Again, assume one call takes A seconds to complete. If you did the exercise as intended, every call causes the function itself to be called *twice*. Thus, the amount of time it will take to complete the calculation will be approximately (assuming n is large) $A \cdot 2^n$. As you can see, this is much slower as n increases.

Two more algrithms you should consider: Our sorting algorithm and the "for loop" implementation of the Fibonacci function.