The Allan Technique of Programming Recursively

Students have difficulty with recursion. I once had a professor who said that recursion would become as natural for me as iteration. Although I doubted him at the time, it has become true. Here are a few steps I have picked up over the years.

1. Decide that the problem needs a recursive solution. Likely candidates are operations on a recursive structure (such as trees) or problems which are defined recursively (like Ackermann's function).

   Some people avoid recursion as it is more costly than iteration. I believe that good compilers will eventually eliminate most of the overhead of recursive calls. Plus, the convenience and maintainability of recursive routines make them a good choice.

2. Decide specifically what the procedure will do. What are its inputs and what are its outputs. In a sentence, what does it accomplish?

3. Restate the problem recursively. Look for instances of the problem you defined in step 2.

   If the procedure returns a value, recursive calls to the function must use the returned values and every path through the function must return a value.

4. Write the procedure using instances of itself. Only worry about a single call of the routine. This is similar to proof by induction. You only solve for one case, but all other cases take care of themselves.

   Do not be tempted to follow the recursion. Use the routines having the faith that they will work. (It is a self-esteem thing. Believe in yourself!) If you try to follow the recursion while you are writing the procedure, you will become hopelessly lost. How can you follow the recursion when the recursive routine is not finished yet?

5. Make sure you take care of the base case stopping the recursion. I like to take care of this last as it is easier to pinpoint the ending conditions after the general case has been written. Some personalities may not be able to delay this decision, but you will find coding is faster if you are able to delay it. It is like delaying the condition on a loop. In all but the simplest of loops, it is easier to write the termination condition after the loop body is written.

6. Once the routine is written, go ahead and follow the recursion, if you wish.

Let's apply these steps to finding the biggest element in an array.

1. This doesn't need to be recursive, but it is a simple example on which to demonstrate double recursion.

2. This problem could be coded as a function having the following prototype. \( \text{FindBig}(s,low,high) \) returns the biggest element of the array \( s \) between subscripts \( low \) and \( high \).

   \[
   \text{T FindBig(T s[], int low, int high)}
   \]

3. One can find the biggest element of an array by finding the biggest element of the two halves and then comparing them.
A first attempt may be:

```c
char FindBig(char s[], int low, int high)
{
    int mid = (low+high)/2;
    char leftBig = FindBig(s, low, mid);
    char rightBig = FindBig(s, mid+1, high);
    return (leftBig>rightBig? leftBig : rightBig);
}
```

The base case is when there is a single element in the array slice.

```c
char FindBig(char s[], int low, int high)
{
    if (low==high) return s[low];
    int mid = (low+high)/2;
    char leftBig = FindBig(s, low, mid);  // Let 100 be address following
    char rightBig = FindBig(s, mid+1, high);  // Let 108 be address following
    return (leftBig>rightBig? leftBig : rightBig);
}
```

```c
void main()
{
    int i;  /* loop index - character of val array to be inserted */
    char *val="ameirxlcedq";
    cout <<  "Biggest is " << FindBig(val,0,strlen(val)-1);  // let 200 be address
}
```

To follow the recursion, you really need a stack in which return addresses are kept. Here is a machine generated trace of the program. The actual program which produced this output is shown below. You may want to play around with it.

```
Input array is: ameirxlcedq
Biggest is
1.    Entering (0:10)
2.        Entering (0:5)
3.                Entering (0:2)
4.                        Entering (0:1)
5.                                Return (0:1)==>m
6.                        Return (0:2)==>m
7.                Entering (3:5)
8.                        Return (3:4)==>r
9.                Return (3:5)==>x
10.         Return (0:5)==>x
11.        Entering (6:10)
12.                Entering (6:8)
13.                        Entering (6:7)
14.                                Return (6:7)==>l
15.                        Return (6:8)==>l
16.                Entering (9:10)
17.                        Return (9:10)==>q
18.                Return (6:10)==>q
19.                Return (0:10)==>x
20.        Return (0:10)==>x
```
char FindBig(string indent, char s[], int low, int high) {
    if (low==high) return s[low];
    cout << indent << "Entering (" << low << ":" << high << ")\n";
    int mid = (low+high)/2;
    char leftBig = FindBig(indent+"    ", s, low, mid);
    char rightBig = FindBig(indent+"    ", s, mid+1, high);
    char Big = leftBig>rightBig? leftBig : rightBig;
    cout << indent<< "Return (" << low << ":" << high << ")==" << Big << endl;
    return Big;
}

void main()
{ char *val="ameirxlcedq"
    cout << "Biggest is " << FindBig("",val,0,strlen(val)-1);
}