Tail Recursion

Recursive functions solve a problem by calling themselves with an ever smaller subset of the data. You normally learn about recursion by calculating the Fibonacci sequence. I will use a more trivial example first; add() sums the numbers from 1 to n recursively.

```
int add(int n)
   {
   if (n==1)
      return 1;
   else
      return (n + add(n-1));      // recurse on next smaller number
   }
```

When you call add(3), for instance, you return from the else statement with 3 + add(2) as a result. Next, add(2) returns 2 + add(1), which then recurses to return 1. All of the steps: 3 + add(2) and 2 + add(1) got pushed down into the stack. When you reach the tail, where n==1, the recursion bottoms out. The return 1 statement pops the stack to each previous level, summing the numbers as it goes.

Here is a recursive function to find the any value of the fibonacci sequence.

```
long int fib(long int x)
  {
  if (x==0)
    return 0;

  else if (x==1)
    return 1;

  else
    return fib(x-1) + fib(x-2);
  }
```

Each time fib() is called with a smaller number, pushing the current value onto the stack. At each level of the recursion you add another value to the stack. When you reach fib(1) you have found the bottom, then you work your way out of the stack, to obtain the result.

Rewrite this function to be iterative instead of recursive, and compare the time it takes each to perform its task. You will find the iterative function is faster, using less memory space. If your recursive function burrows too deeply, it can use all of the stack space available to your process and crash. This is why you want to minimize your use of tail recursion if possible. Modern compilers do optimize code to fix the problem, but it is not wise to depend on your optimization routines.

Here is an example where I built a tree data structure recursively. Trees and recursion work well together. However, each time you recurse down a level you are using stack memory. Luckily each time you step down a level in a tree you are cutting the problem in half, or in this case into quarters.

Using recursion for this problem saves time in writing the code, while using a little more time when running it. Though I could have written the function iteratively it would be less concise and harder to read. This code is clear to me, even if it does use a little longer to get the job done.

```
void qTree::walkQuadTree(qTree *here)
   {
   if (here->sw != NULL)                          //  -,- quadrant
      walkQuadTree(here->sw);
   if (here->nw != NULL)                          //  -,+ quadrant
      walkQuadTree(here->nw);

   // do node work here

   if (here->se != NULL)                          //  +,- quadrant
      walkQuadTree(here->se);
   if (here->ne != NULL)                          //  +,+ quadrant
      walkQuadTree(here->ne);
   return;
   }
```

If you use recursion wisely, it can be helpful and keep your code compact. Monitor your use of stack space. If you find the code running slowly, you may want to convert your recursive code to an iterative function instead. But, with modern CPUs being quite fast, you may not notice any problems. I think the value of clean and concise code outweighs the downside, in most cases.

https://www.geeksforgeeks.org/tail-recursion/

https://cs.stackexchange.com/questions/6230/what-is-tail-recursion

https://www.quora.com/What-is-tail-recursion-Why-is-it-so-bad

https://stackoverflow.com/questions/33923/what-is-tail-recursion