159.271 Computational Thinking for Problem Solving

4. Recursion Revisited

4.1 How to Think About Recursion

Back to top

There are a few different ways to view a recursive algorithm. The easiest method is to focus on one step at a time. Suppose that someone gives you an instance of a computational problem. You solve it this way: if it is pretty small, solve it yourself, if not, get a number of friends to help you. You construct for each friend an instance of the same computational problem that is *smaller* than your own. We refer to these as *subinstances*. Note that each friend gets a *different* instance of the same problem that you start with. Your friends will kindly provide you with the solutions to these subinstances. You then combine these subsolutions into a solution for your original instance.

The key is that you concern yourself only with your own task. Don't worry about how your friends solve the subinstances that you assigned them. Similarly, don't worry about whoever gave you your instance and what he does with your answer. Trust your friends!

Consider a row of houses. Each house bigger than the next. Your task is to get into the biggest house. You are locked out of all the houses. The key to each house is locked in the house of next smaller size. The recursive problem consists of getting into any specified house. Each house in the row is a separate *instance* of this problem.

Suppose that the smallest house is so small that we can just lift the roof off. Once in this house we can get the key to the next house, which is then easily opened. Within this house, we find the key to the next house after that, and so on. Eventually, we are in the largest house as required.

Working forward vs. backward. An iterative algorithm works forwards. It knows about house i-1. It uses a loop invariant to show that this house has been opened. It searches this house and finds the key to house i. Because of this, it decides that house i would be a good one to go to next.

A recursive algorithm works backwards. It knows about house i. It wants to get it open. It determines that the key for house i is in house i-1. Therefore, opening house i-1 is a *subtask* that needs to be accomplished. There are two advantages here. The first is that sometimes it is easier to work backwards. The second is that a recursive algorithm is allowed to have more than one subtask involved. This forms a *tree* of houses to open instead of a row of houses.

Steps to follow when developing a recursive algorithm

1. **Specifications:**

   Carefully write the specifications for the problem. **`Preconditions`** state any assumptions that must be true about the input instance for the algorithm to operate correctly. **`Postconditions`** are statements about the output that must be true when the algorithm returns. This step is even more important for recursive algorithms, because there must be tight agreement between what is expected from you in terms of pre and post-conditions and what is expected from your friends.

   Both the pre and post-conditions act as **`loop invariants`**. The loop invariant in an iterative algorithm states what is maintained as the control gets passed from iteration to iteration. It provides a picture of what you want to be true in the middle of the computation. With recursion, however, there are two directions. The precondition states what you want to be true halfway down the recursion tree. The postcondition states what you want to be true halfway back up the recursion tree.

2. **Variables**

   With recursive programs there are variables that play specific roles and that should be used in specific ways.

   - Your Input: Your mission, if you choose to accept it, is received through your input parameters. The first line of your code **`Alg((a, b, c))`** specifies both the name of the routine and the names of its parameters. Here, **`(a, b, c)`** is the input instance that you need to find a solution for.
   - Your output: You must return a solution **`(x, y, z)`** to your instance **`<(a, b, c)`** through a return statement **`return((x, y, z))`**. If your code has **`if`** or **`loop`** statements, then every path through the code must end with a return statement. Each return statement must return a solution **`(x, y, z)`** of the right type.
   - Your friend's input: You must create a subinstance $(a_{sub}, b_{sub}, c_{sub})$ for each friend. You pass this new set of inputs to a friend by recursing with $\texttt{Alg}((a_{sub}, b_{sub}, c_{sub}))$. Each subinstance must meet the preconditions of your problem.
   - Your friend's output: Trust that each friend will return a correct solution $(x_{sub}, y_{sub}, z_{sub})$ to the subinstance $(a_{sub}, b_{sub}, c_{sub})$. Make sure to save each result in variables of the correct type, using $(x_{sub}, y_{sub}, z_{sub}) = \texttt{Alg}((a_{sub}, b_{sub}, c_{sub}))$

3. **Tasks to Complete**

   Accept your mission. Know the range of things that your input instance might be. Suppose the inout instance is a binary tree. Make sure that your program works

for a general tree with non-empty left and right subtrees, a tree with a non-empty left subtree and empty right subtree, a tree with a non-empty right subtree and empty left subtree, and the empty tree.
    - Construct subinstances. Make sure that these meet the preconditions of your problem. Make sure that the subinstances that you construct are *smaller* than your own instance.
    - Trust your friend. Focus only on your mission. Trust your friend to give you a correct solution to the instance that you give her. Don't think further down the line than that. Your job is only to figure out the right instance to give her in the first place.
    - Construct your solution. Using the solutions returned by your friends for your subinstances, your next task is to construct a solution for your instance. This generally requires a block of code, sometimes just a single line will do. Make sure that your algorithm **`returns`** the solution that you construct.
    - Base cases. Consider which instances get solved by your program. For those that don't, add base cases to solve them in a brute force way. If your input instance is sufficiently small, then you must solve it yourself as a base case.

The Stack Frame

Tree of Stack Frames. To have some picture of the entire computation for a recursive algorithm, the tree of stack frames level of abstraction is best. The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance. A single routine can have many executions going on a the "same" time . Each such execution is referred to as a stack frame.

If each routine makes a number of subroutine calls (recursive or not), then the stack frames that get executed form a tree.

Stack of Stack Frames. The algorithm is actually implemented on a computer by a stack of stack frames. What is stored in the computer memory at any given point in time is only a single path down the tree.

Memory. The routine itself is described only once, by a block of code that appears in static memory. This code declares a set of variables. Each instance of this routine that is currently being executed may be storing different values in these variables and so needs to have its own separate copy of these variables. The memory requirements of each of these instances are stored in a separate stack frame.

Using a Stack of Stack Frames. Let us denote the top stack frame by A. When the execution of A makes a subroutine call to a routine with some input values, a stack frame is created for this new instance. This frame B is pushed onto the stack after that for A. In addition to a separate copy of the local variables for the routine, it contains a pointer to the next line of code that A must execute when B returns. When B returns, its stack frame is popped and A continues to execute at the line of code indicated by the stack frame B. When A completes, it too is popped off the stack.

4.2 Running Times for Recursive Algorithms

Back to top

Consider the classic recursive sorting algorithm **`MergeSort`**. We recursively give one friend the first half of the input list to sort and another friend the second half to sort. Then we combine these two sorted sublists into one completely sorted list.

If we think about what happens at one recursion step of the algorithm above, we realise that we have taken a problem that is of size **`N`** and swapped it for two problems of size **`N/2`**. The time that a size **`N`** problem takes is **`T(N)`**, where this is some sort of **`O()`** function, but we don't know what. This is known as a *recurrence relation* that links the running time of the different problems. For this problem we have:

```
T(N) = 2T(N/2) + O(N)
```

No matter what the size of the problem **`N`** is, where the **`O(N)`** part is the cost of splitting the data into two and adding the solutions up afterwards, at each step down the recursion tree we have halved the size of the problem. So how many possible levels of the recursion tree are there? We halve the size of the data array each time, and therefore there are at most $\log_2(N)$ levels in the tree. This leads to computational complexity **`O(N log N)`**, where we always take **`log`** to be $\log_2$.

In general, if a routine recurses **`a`** times on instances of size **`n/b`** , then the related recurrence relation is **`T(n) = a . T(n/b) + f(n)`**. If the routine recurses **`a`** times on instances of size **`n - b`** then the recurrence relation will be **`T(n) = a . T(n - b) + f(n)`**.

Working out these recurrence relations for recursive algorithms is fairly easy. What isn't necessarily easy is converting them into complexities. So here is a handy cut-out-and-keep guide:

If $T(n) = a \,.\, T(n/b) + O(n^d)$, where **`a, b, d`** are constants, then:

$$\text{complexity} = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d.\log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

If **`T(n) = a . T(n - b) + f(n)`**, where **`a, b, d`** are constants, then:

$$\text{complexity} = \begin{cases} O(a^{n/b}) & \text{if } a > 1 \\ O(n.\,f(n)) & \text{if } a = 1 \end{cases}$$

Intuitively, the number of subinstances grows exponentially by a factor of **`a`**. On the other hand, the sizes of the subinstances shrink exponentially by a factor of **`b`**. The amount of the work that the instance must do is a the function **f** of this instance size. Whether the growing or the shrinking dominates this process depends on the relationship between **a**, **b**, and **f(n)**.

Look back at the recursive algorithm for the Fibonacci numbers that was discussed in Chapter 3. The recurrence relation for the program is **T(n) = T(n-1) + T(n-2)**. This is just the same as the formula for the Fibonacci numbers, so the computational time grows with the Fibonacci number themselves, not very useful. Have a look at the second and third algorithms. They consist of a single loop that does one computation, and that runs **n-1** times, so the algorithm is linear in **n**, which explains why they run so well.

4.3 Recursion on Trees

Back to top

A Tree is naturally a recursive data structure, as is a linked list. These sorts of data structures naturally lend themselves to recursive algorithms. You will be familiar with recursive algorithms to perform tree traversals, in the three classic orders *prefix, infix* and *postfix*.

Let us develop an algorithm to decide whether or not a given tree is a binary search tree. Recall that in a binary search tree the nodes are ordered so that, for each node, all the keys in its left subtree are smaller than its key and all the keys in its right subtree are larger.

We will see in this case, that it is easier to recursively solve a more general version of this problem, either by providing more information about the original instance, or by asking for more information about subinstances.

```
algorithm IsBSTtree(tree):
        # Precondition: tree is a binary tree
        # Postcondition: Output indicates whether or not tree is a
BST

        begin:
          if (tree == emptyTree):
             return YES
          elsif IsBSTtree(tree.leftSub) and IsBSTtree(tree.rightSub)
                and Max(tree.leftSub) <= rootKey(tree) <=
Min(tree.rightSub):
             return YES
          else:
             return NO
         end
```

For each node in the input tree, this algorithm needs to compute the minimum or maximum value in the node's left and right subtrees. If the input tree is unbalanced, say, a single path, then for a node i, computing either the minimum or maximum of its (single) subtree will involve traversing to the bottom of the path. Therefore, the running time of this algorithm will be $O(n^2)$.

It is better to combine the **IsBSTtree** and **Min** and **Max** routines into one routine so that the tree only needs to be traversed once. We do this by returning more information about subinstances. In addition to whether or not the tree is a binary search tree, the routine will return the minimum and maximum value in the tree. If our instance is the empty tree then we return a special value that is greater than all keys in the tree as the minimum and a special value that is less than all the keys as a maximum.

```
algorithm IsBSTtree(tree):
        # Precondition: tree is a binary tree
        # Postcondition: Output indicates whether or not tree is a
BST
        #                also gives min and max values in the tree

        begin:
          if (tree == emptyTree):
             return (YES, +infinity, -infinity)
          else:
             (leftIs, leftMin, leftMax) = (IsBSTtree(tree.leftSub))
             (rightIs, rightMin, rightMax) =
(IsBSTtree(tree.rightSub))
             min = Min(leftMin, rightMin, rootKey(tree))
             max = Max(leftMax, rightMax, rootKey(tree))
             if leftIs and rightIs and leftMax <= rootKey(tree) <=
rightMin:
                isBST = YES
             else:
                isBST = NO
             return (IsBST, min, max)
         end
```

Another algorithm for the IsBST problem generalizes the problem by providing more information via the input. The more general problem, in addition to the tree, will provide a range of values `[max, min]` and ask whether or not the tree is a BST with values within this range. We start by calling `IsBSTtree(tree, [-infinity, +infinity])`.

```
algorithm IsBSTtree(tree):
        # Precondition: tree is a binary tree, [min, max] is a range
of values
        # Postcondition: Output indicates whether or not tree is a
BST with values within [min, max] range

        begin:
          if (tree == emptyTree):
             return YES
          elsif rootKey(tree) in [min, max] and
                IsBSTtree(tree.leftSub, [min, rootKey(tree)]) and
                IsBSTtree(tree.rightSub, [rootKey(tree), max]):
             return YES
          else:
             return NO
         end
```

4.4 Practice Questions

Back to top

1. Question 4.1 - Develop an algorithm to compute the number of leaves in a binary tree.
2. Question 4.2 - Write a recursive program that takes a BST and an integer k as input and returns the kth smallest element in the tree.
3. Question 4.3 - Consider the algorithm below, what will the tree of stack frames look like

for this algorithm on input **(2,3)**?

```
algorithm A(k,n):
          if (k == 0):
             return(n+1+1)
          elsif (n == 0):
             if (k == 1):
                return(0)
             else:
                return(1)
          else:
             return(A(k-1, A(k, n-1)))
```

4. Question 4.4 - The Towers of Hanoi. The classic version of the problem says that you have 5 disks of different sizes stacked in size order on the first of three poles. The problem is to move the set of disks onto the last of the three poles by moving one disk at a time in such a way that you never put a larger disk on top of a smaller one. Complete the Towers of Hanoi problem and implement your algorithm in Python.