Justin Ciocoi

Oct. 11, 2023

CSCI 373 Textbook Notes

Recursion

- *Repetition* is a key feature of high level languages, and we have seen that repetition can be achieved through for and while loops
- Another way to achieve repetition is through **recursion**, which occurs whenever a function calls itself within its own definition
- **The Factorial Function**
  - Let us define the factorial function
    - $n! = 1$, if $n = 0$
    - $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$ if $n \geq 1$
  - For example,
    - $5! = 5 * 4 * 3 * 2 * 1 = 120$
  - From this, we can see,
    - $5! = 5 * (4!)$
    - and then, since $4! = 4 * 3 * 2 * 1$
      - $5! = 5 * 4 * (3!)$
  - So this leads to the following recursive definition of the recursive factorial function
    - factorial(n) $= 1$, if $n = 0$
    - factorial(n) $= n \cdot factorial(n-1)$, if $n \geq 1$
  - As we can see, this function contains one or more **base cases**
    - In this case, the base case is 1 when $n = 0$

- There is no circularity in this definition because each time the function is invoked, its argument is smaller by one

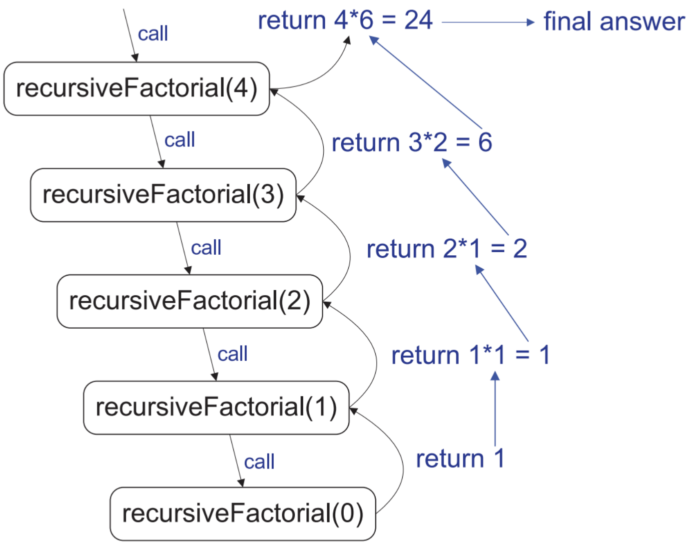
- **C++ Implementation of Recursion in the Factorial Function**
  - Note that in the following definition, no loops are necessary since recursion is used
  - ```
    int recursiveFactorial(int n)
    {
        if(n==0)
            return 1; //basis case which will always be called at the end
        else
            return n * recursiveFactorial(n-1); //recursive case
    }
    ```
  - This function can be illustrated using the following recursion trace
  - 

    Figure 3.16: A recursion trace for the call recursiveFactorial(4).

- **Recursive Example using an English Ruler**
  - An English ruler is broken into intervals and each interval contains a set of *ticks*
  - These ticks are placed at intervals of $\frac{1}{2}$ inch, $\frac{1}{4}$ inch, and so on
  - As the size interval decreases by half, the tick length decreases by one
  - Below are some representations of English Rulers
  -
    ```

```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1          ----            --- 2
-               -               -
--              --              --
-               -               -
---             ---             --- 3
-               -
--              --
-               -
---- 2          ----- 1

(a)             (b)             (c)
```

Figure 3.17: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

- The longest tick length of an English Ruler will be referred to as the *major tick length*
- One approach to drawing this consists of three functions
  - `drawRuler()` draws the entire ruler and takes the number of inches, `nInches`, and the major tick length, `majorLength` as arguments
  - The utility function, `drawOneTick()`, draws a single tick of the given length
  - `drawTicks`, which is the recursive function which draws the sequence of ticks within some interval
- Here is a C++ implementation of what is described above
- ```
  void drawOneTick(int tickLength, int tickLabel = -1)
  {
      for(int i=0; i<tickLength; i++)
          cout<<"-";
      if(tickLabel>=0)
          cout<<" "<<tickLabel;
      cout<<endl;
  }
  ```
- This function draws one tick with an optional label
  ```

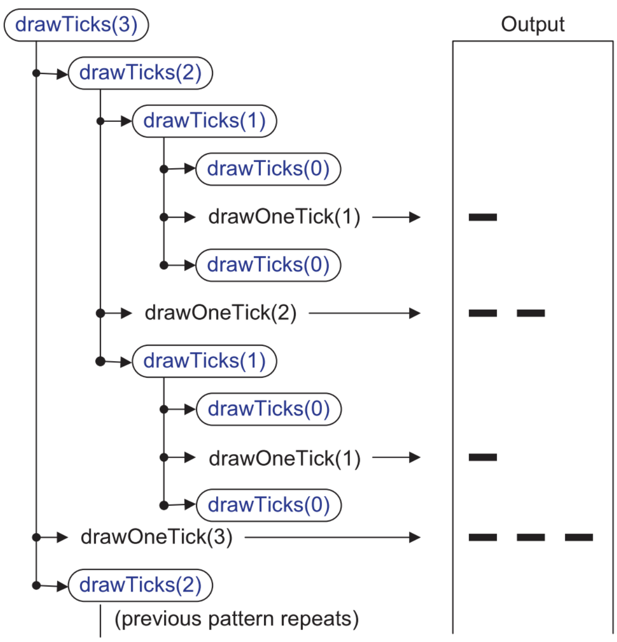
- ```
  void drawTicks(int tickLength)
  {
      if(tickLength>0)
      {
          drawTicks(tickLength-1);
          drawOneTick(tickLength);
          drawTicks(tickLength-1);
      }
  }
  ```
- This function recursively draws ticks between two major ticks
- ```
  void drawRuler(int nInches, int majorLength)
  {
      drawOneTick(majorLength, 0);
      for(int i=1; i<nInches; i++)
      {
          drawTicks(majorLength-1);
          drawOneTick(majorLength, i)
      }
  }
  ```
- This function can be used to draw the ruler as a whole
- 

  Figure 3.18: A partial recursion trace for the call drawTicks(3). The second pattern of calls for drawTicks(2) is not shown, but it is identical to the first.
- The above recursion trace provides an illustration of what will occur when `drawRuler` is run with a major tick length of 3

- **More Examples of Recursion**
  - Recursion can be beneficial by allowing us to exploit a more *natural* form of repetition that does not involve complex nested loops or case analyses
  - *Example 3.1:* Modern OSes operate file-system directories in a recursive manner, meaning folders can be nested inside of folders in an arbitrarily deep fashion so long as there is sufficient space in memory
  - *Example 3.2:* The syntax in modern programming languages is most often defined in a recursive manner
- **3.5.1: Linear Recursion**
  - Linear recursion is the simplest form of recursion
  - Linear recursion refers to a recursive function that makes, at most, one recursive call each time that it is invoked
  - *Summing the elements of an array recursively*
    - Suppose we have an array, A, of n integers which we want to sum together
    - Since we know that the sum of all integers in A is equal to $A[0]$ when $n = 1$, we can solve this problem recursively with the following algorithm
      - ```
        Algorithm for LinearSum(A, n)

            Input: an integer Array, A, and int n >= 1
            Output: The sum of the first n integers in A

            if n=1 then
                return A[0]
            else
                return LinearSum(A, n-1)+A[n-1]
        ```
    - This illustrates one very important aspect of *all* recursive functions - the fact that it terminates
      - This can be fairly easily achieved by writing a non recursive statement for the base case, in this case the `if n=1` statement achieves this
  - In fact, an algorithm that employs linear recursion generally adheres to the following form
        ```

- *Test for base cases*, where the function reaches a pre-defined base case for which a recursive call is not needed
    - Base cases should be defined such that every possible chain of recursive calls eventually reaches a base case
- *Recursion*, where after testing for base cases, the function will recursively call itself
    - It might have to decide between different recursive steps, but a linear recursive algorithm should call itself recursively only once each time it is invoked

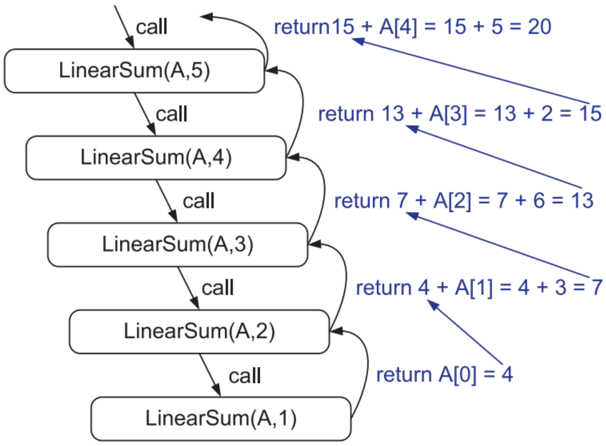
- Now, let us consider the recursion trace, or visual diagram representing a system's logic during a recursive linear summation
- 

  Figure 3.19: Recursion trace for an execution of LinearSum(A, n) with input parameters $A = \{4, 3, 6, 2, 5\}$ and $n = 5$.
- Recursive algorithms can, however, take up more space in the memory due to their need to store each prior recursive call until the function terminates
- Therefore, it can sometimes be useful to be able to derive non-recursive algorithms from recursive ones
- *Tail recursion* occurs when a recursive algorithms initiates the recursive call as the last thing it does other than base case evaluation
- Algorithms that utilize tail recursion are simple to convert from recursive to non-recursive
    - This can be achieved by iterating through the recursive calls rather than calling them explicitly
- Here is the algorithm for `IterativeReverseArray()`

- ```
  Algorithm: IterativeReverseArray(A, i, j)

      Input: An array A, and non-negative integer indices i and j
      Output: Reversal of A from i to j

      while i<j
          Swap A[i] and A[j]
          i <- i+1
          j <- j-1
      return
  ```

- 3.5.2: Binary Recursion
  - When a function makes two recursive calls, we can refer to this as *binary recursion*
  - Let us look at the algorithm for a Binary sum
  - ```
    Algorithm BinarySum(A, i, n)

        Input: An array A an integers i and n
        Output: The sum of the n integers in A starting at i

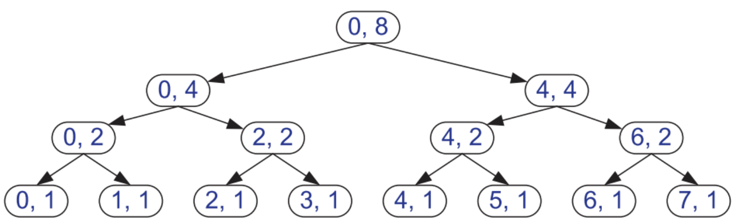
        if n=1
            return A[i]
        return
            BinarySum(A, i, [n/2]) + BinarySum(A, i+[n/2], [n/2])
    ```
  - Below is the trace for Binary Sum
    - 

      Figure 3.20: Recursion trace for the execution of BinarySum(0,8).

- Computing Fibonacci Numbers via Binary Recursion

- ```
  Algorithm BinaryFib(k):

      Input: Nonnegative integer k
      Output: The kth Fibonacci number Fk

      if k ≤ 1 then
          return k
      else
          return BinaryFib(k-1) + BinaryFib(k-2)
  ```
- However, this is of time complexity $O(n^2)$, because the number of recursive calls more than doubles with each consecutive index
- Therefore, it is actually more efficient to compute the k_{th} Fibonacci number using *linear recursion*

- **Computing Fibonacci Numbers via Linear Recursion**
  - ```
    Algorithm LinearFibonacci(k):

        Input: A nonnegative integer k
        Output: Pair of Fibonacci numbers (Fk,Fk-1)

        if k ≤ 1 then
            return (k,0)
        else
            (i, j)← LinearFibonacci(k-1)
            return (i+ j, i)
    ```
  - For this algorithm, the time complexity is $O(n)$ so it is far more efficient than binary recursion for Fibonacci calculations
- **3.5.3: Multiple Recursion**
  - If we generalize the jump from linear to binary recursion, we can arrive at *multiple recursion*
    - Multiple recursion algorithms may make multiple recursive calls, with that number being possibly more than two
  - Below is the algorithm and recursion trace for an algorithm written to solve *summation puzzles* where different letters represent integers in an equation

■

```
Algorithm PuzzleSolve(k,S,U):

    Input: An integer k, sequence S, and set U
    Output: An enumeration of all k-length extensions to S using elements
in U
            without repetitions

    for each e in U do
        Remove e from U {e is now being used} Add e to the end of S
        if k = 1 then
            Test whether S is a configuration that solves the puzzle
            if S solves the puzzle then
                return “Solution found: ” S
        else
            PuzzleSolve(k-1,S,U)
        Add e back to U {e is now unused}
        Remove e from the end of S
```

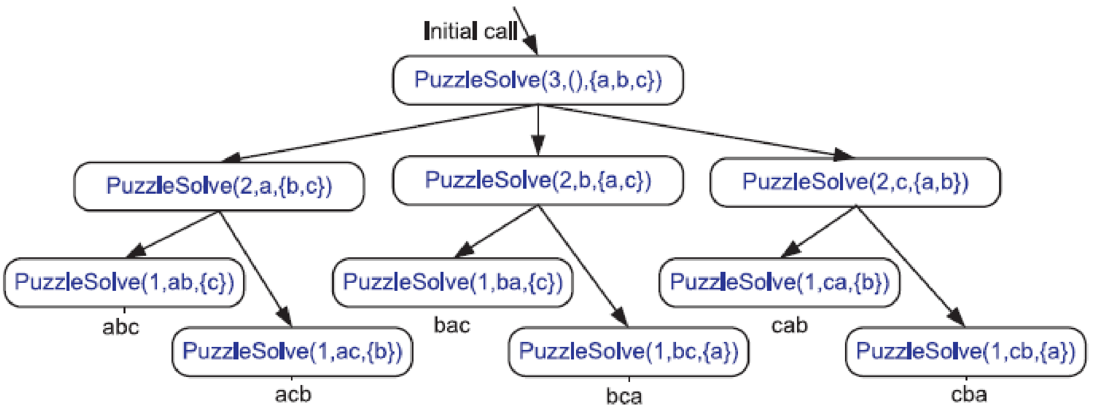


Figure 3.21: Recursion trace for an execution of PuzzleSolve$(3,S,U)$, where S is empty and $U = \{a,b,c\}$. This execution generates and tests all permutations of a, b, and c. We show the permutations generated directly below their respective boxes.

■