Introduction to Recursion

1 Recursion

Recursion is a powerful tool for solving certain kinds of problems. Recursion breaks a problem into smaller problems that are identical to the original, in such a way that solving the smaller problems provides a solution to the larger one.

It can be used to define mathematical functions, languages and sets, and algorithms or programming language functions. It has been established that these are essentially equivalent in the following sense: the set of all functions that can be computed by algorithms, given some reasonable notion of what an algorithm is, is the same as the set of all functions that can be defined recursively, and each set (or language) for which membership can be determined by an algorithm corresponds to a function that can be defined recursively.

We are interested here mostly in the concepts of recursive algorithms and recursion in programming languages, but we also informally introduce recursive definitions of functions.

2 Recursive Algorithms

2.1 Example: The Dictionary Search Problem

Suppose we are given a problem to find a word in a dictionary. This is known as a **dictionary search**. Suppose the word is "yeoman". You could start at the first page and look for it and then try the second page. then the third, and so on, until finally you reach the page that contains the word. This is called a **sequential search**. Of course no one in their right mind would do this because everyone knows that dictionaries are sorted alphabetically, and that therefore there is a faster way that takes advantage of the fact that they are sorted. A **dictionary** by definition has the property that the words are listed in it in alphabetical order, which in computer science means it is a sorted list.

One more efficient solution might be to use **binary search**, which is described by the following *recursive algorithm*:

```
1 if ( the set of pages is just one page )
2     scan the page for the word
3 else {
4     open the dictionary to the page halfway between the first and last pages
5     of the set of pages being searched;
6     compare the word to the word at the top of the page;
7     if ( the word precedes the index word on the page alphabetically )
8         search the lower half of the set of pages using this same algorithm
9     else
10        search the upper half of the set of pages using this same algorithm
11  }
```

The recursion in this algorithm occurs in lines 8 and 10, in which the instructions state that we must use this algorithm again, but on a smaller set of pages. The algorithm basically reduces the problem to one in which it compares the word being sought to a single word, and if it is "smaller", it looks for the word in the first half, and if it is larger, it looks in the second half. When it does this "looking again", it repeats this exact logic. This approach to solving a problem by dividing it into smaller identical problems and using the results of conquering the smaller problems to conquer the large one is called **divide-and-conquer**. Divide-and-conquer is a problem-solving **paradigm**, meaning it is a model for solving many different types of problems.

The binary search algorithm will eventually stop because each time it checks to see how many pages are in the set, and if the set contains just one page, it does not do the "recursive part" but instead scans the page, which takes a finite amount of time. It is easier to see this if we write it as a pseudo-code function:

```
void  binary_search( dictionary_type dictionary,  word keyword ) {
    if ( the set of pages in dictionary has size = 1  ) {
        // This is called the base case
        scan the page for keyword;
        if ( keyword is on the page )
            print keyword's definition;
        else
            print "not in dictionary";
    }
    else { // the size > 1
        let middlepage be the page midway between the first and last pages
        of the set of pages being searched; // e.g., middlepage=(first+last)/2
        compare keyword to indexword at the top of middlepage;
        if ( keyword < indexword )
            // recursive invocation of function
            binary_search( lower half of dictionary, keyword );
        else
            // recursive invocation of function
            binary_search( upper half of dictionary including middlepage, keyword );
    }
}
```

Observations

1. This function calls itself recursively in two places.
2. When it calls itself recursively, the size of the set of pages passed as an argument is at most one-half the size of the original set.
3. When the size of the set is 1, the function terminates without making a recursive call. This is called the base case of the recursion.
4. Since each call either results in a recursive call on a smaller set or it terminates without making a recursive call, the function must eventually terminate in the base case code.
5. If the keyword is on the page checked in the base case, the algorithm will print its definition, otherwise it will say it is not there. This is not a formal proof that it is correct!

If a recursive function has these two properties:

- that each of its recursive calls diminishes the problem size, and
- that the function takes a finite number of steps when the problem size is less than some fixed constant size,

then it is guaranteed to terminate eventually. Later we will see a more general rule for guaranteeing that a recursive function terminates.

2.2 Example: The Factorial Function

Although the factorial function ($n!$) can be computed by a simple iterative algorithm, it is a good example of a function that can be computed by a simple recursive algorithm. As a reminder, for any positive integer n, *factorial(n)*, written mathematically as $n!$ is the product of all positive integers less than or equal to n. It is often defined by $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1$. The "..." is called an **ellipsis**. It is a way of avoiding writing what we really mean because it is impossible to write what we really mean, since the number of numbers between $(n-2)$ and 2 depends on the value of n. The reader and the author both agree that the ellipsis means "and so on" without worrying about exactly what “and so on” really means.

From the definition of $n!$ we have

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1$$

and

$$(n-1)! = (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1$$

By substituting the left-hand side of the second equation into the right-hand side of the first, we get

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

This would be a circular definition if we did not create some kind of stopping condition for the application of the definition. In other words, if we needed to find the value of 10!, we could expand it to $10 \cdot 9!$ and then $10 \cdot 9 \cdot 8!$ and then$10 \cdot 9 \cdot 8 \cdot 7!$ and so on, but if we do not define what 0! is, it will remain undefined. Hence, this circularity is removed by defining the base case, $0! = 1$.

The definition then becomes:

$$n! \quad = \quad \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1)! & if\ n > 0 \end{cases}$$

This is a recursive definition of the factorial function. It can be used to find the value of $n!$ for any non-negative number n, and it leads naturally to a recursive algorithm for computing $n!$, which is written in C below.

```
int factorial( int n)

    /*
    Precondition:  n >= 0
    Postcondition: returns  n!
    */
    {
        if ( 0 == n)
            return 1;
        else
            return n * factorial(n-1);
    }
```

Observations

1. This does not result in an infinite sequence of calls because eventually the value passed to the argument of factorial is 0, if it is called with $n >= 0$, because if you "unwind" the recursion, you see each successive call is given an argument 1 less than the preceding call. When the argument is 0, it returns a 1, which is the base case and stops the recursion.
2. This function does not really compute $n!$ because on any computer, the number of bits to hold an `int` is always finite, and for large enough n, the value of $n!$ will exceed that largest storable integer. For example; 13! = 6,227,020,800 which is larger than the largest `int` storable on a 32-bit computer.

Food For Thought

There are elementary functions of the non-negative integers that are so simple that they do not need to be defined recursively. Two examples are

$$\begin{aligned} n(x) &= 0 \\ s(x) &= x + 1 \end{aligned}$$

The first has the value zero for all numbers, and the second is the successor of the number. If we allow only two methods of defining new functions, recursion and function composition, what functions can we define?

Consider this function, defined recursively:

$$\begin{aligned} a(x, 0) &= x \\ a(x, y + 1) &= s(a(x, y)) = a(x, y) + 1 \end{aligned}$$

What does it compute?

3 Tracing Recursion: The Box Method

It is hard to trace how a recursive function works. You have to be systematic about it. There are a few established systems for doing this. One is called the **box method**. It is a visual way to trace a recursive function call.

The box method is a way to organize a trace of a recursive function. Each call is represented by a box containing the value parameters of the function, the local variables of the function, a place to store the return value of the function if it has a return value, placeholders for the return values of the recursive calls (if any), and a place for the return address. The steps are as follows.

Steps

1. Label each recursive call in the function (e.g., with labels like 1,2,3,... or A,B,C,...). When a recursive call terminates, control returns to the instruction immediately following one of the labeled points. If it returns a value, that value is used in place of the function call.
2. Create a box template, containing
   (a) a placeholder for the value parameters of the parameter list,
   (b) a placeholder for the local variables,

   (c) a placeholder for each value returned by the recursive calls from the current call,
   (d) a placeholder for the value returned by the function call.

3. Now you start simulating the function's execution. Write the instruction that calls the recursive function with the given arguments. For example, it might be

   cout << factorial(3);

4. Using your box template, create a box for the first call to the function. Draw an arrow from the instruction you wrote in step 3 to this first box.
5. Execute the function by hand, updating values of local variables and reference parameters as needed. For each recursive call that the function makes, create a new box for that call, with an arrow from the old box to the box for the called function. Label the arrow with the label of the function being called.
6. Repeat step 5 for each new box.
7. Each time a function exits, if it has a return value, update the value in the box that called it, i.e., the one on the source side of the arrow, and then cross off the exiting box. Resume execution of the box that called the function at the instruction immediately following the label of the arrow.

We can trace the factorial function with this method to demonstrate the method. The factorial function has a single value parameter, n, and no local variables. It has a return value and a single recursive call. Its box should be

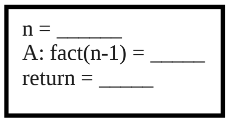

Figure 1: Factorial Box Template

There are three placeholders, one for n, one for the return value of the recursive call, which is labeled A, and one for the return value of the function. The name of the function is abbreviated.

We trace the function for the call when the argument is 3. The figure below illustrates the sequence of boxes. Each row represents a new step in the trace. The first row shows the initial box. The value of n is 3. The other values are unknown. The function is called recursively so the next line shows two boxes. The box in bold is the one being traced. In that box, n=2, since it was called with n-1. That function calls factorial again, so in the next line there are three boxes. Eventually n becomes 0 in the fourth line. It does not make a recursive call. Instead it returns 1 and the box is deleted in the next line and the 1 sent to the fact(n-1) placeholder of the box that called it. This continues until the return values make their way back to the first box, which returns it to the calling program.

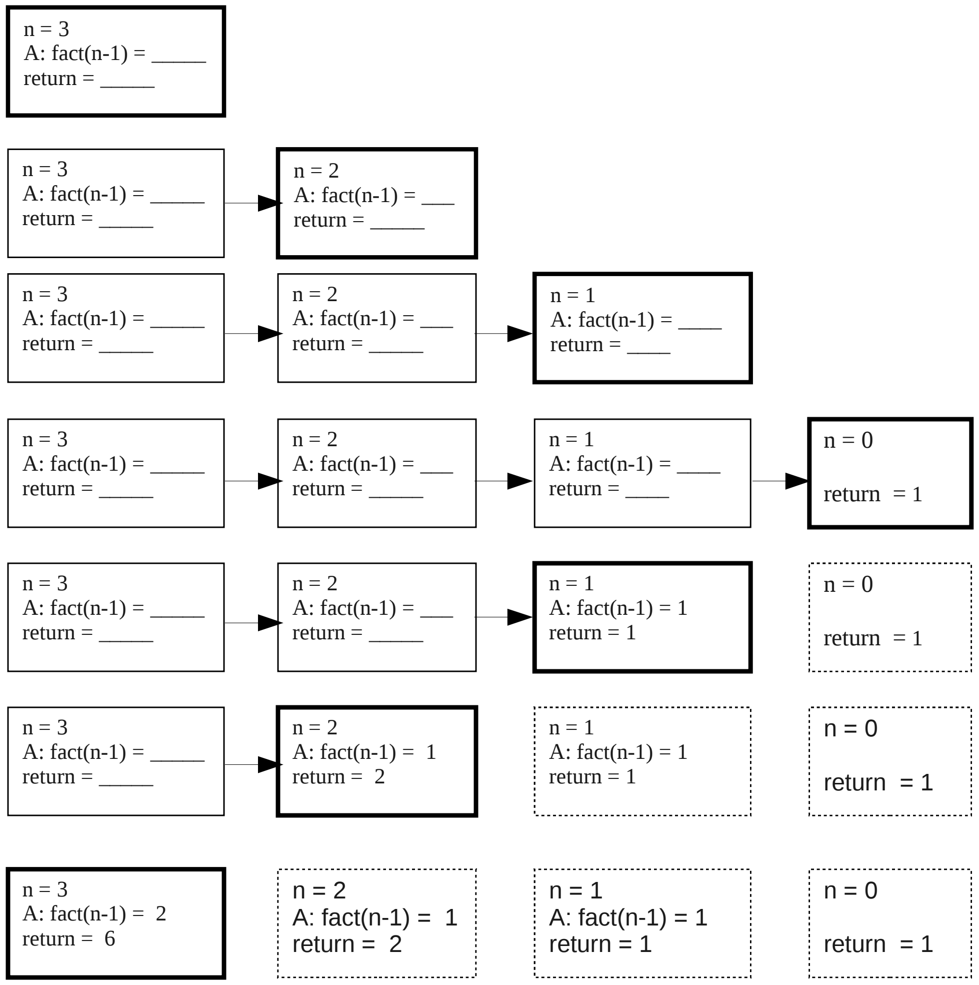

Figure 2: Box Trace of Factorial Function

4 Other Examples

4.1 Fibonacci Numbers

Recursion is not usually the most efficient solution, although it is usually the easiest to understand. One example of this is the Fibonacci sequence. The Fibonacci numbers are named after Leonardo de Pisa, who was known as Fibonacci. He did a population study of rabbits in which he simplified how they mated and how their population grew. In short, the idea is that rabbits never die, and they can mate starting at two months old, and that at the start of each month every rabbit pair gives birth to a male and a female (with very short gestation period!)

From these premises, it is not hard to show that the number of rabbit pairs in month 1 is 1, in month 2 is also 1 (since they are too young to mate), and in month 2, 2, since the pair mated and gave birth to a new pair. Let$f(n)$ be the number of rabbits alive in month n. Then, in month n, where $n > 2$, the number of pairs must be the number of pairs alive in month $n-1$, plus the number of new offspring born at the start of month n. All pairs alive in month $n-2$ contribute their pair in month n, so there are$f(n-1)+f(n-2)$ rabbit pairs alive in month n. The recursive definition of this sequence is thus

$$f(n) = \begin{cases} 1 & if\ n \leq 2 \\ f(n-1)+f(n-2) & if\ n > 2 \end{cases}$$

This will generate the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. A recursive algorithm to compute the nth Fibonacci number, for $n > 0$, is given below, written as a C/C++ function.

```
int fibonacci (int n)
{
    if ( n <= 2 )
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Although this looks simple, it is very inefficient. If you write out a box trace of this function you will see that it leads to roughly 2^n function calls to find the n^{th} Fibonacci number. This is because it computes many values needlessly. There are far better ways to compute these numbers.

4.2 Choosing k Out of n Objects

A well-known and common combinatorial (counting) problem is how many distinct ways there are to pick k objects from a collection of n distinct objects. For example, if I want to pick 10 students in the class of 30, how many different sets of 10 students can I pick? I do not care about the order of their names, just who is in the set. Let $c(n,k)$ represent the number of distinct sets of k objects out of a collection of n objects.

The solution can be difficult to find with a straight-forward attack, but a recursive solution is quite simple. Let me rephrase the problem using the students in the class. Suppose I single out one student, say student X. Then there are two possibilities: either X is in the group I choose or X is not in the group.

How many solutions are there with X in the group? Since X is in the group, I need to pick $k-1$ other students from the remaining $n-1$ students in the class. Therefore, there are $c(n-1,k-1)$ sets that contain student X.

What about those that do not contain X? I need to pick k students out of the remaining $n-1$ students in the class, so there are $c(n-1,k)$ sets.

It follows that

$$c(n,k) = c(n-1,k-1) + c(n-1,k)$$

when n is large enough. Of course there are no ways to form groups of size k if $k > n$, so $c(n,k) = 0$ if $k > n$. If $k = n$, then there is only one possible group, namely the whole class, so $c(n,k) = 1$ if $k = n$. And if $k = 0$, then there is just a single group consisting of no students, so $c(n,k) = 1$ when $k = 0$. In all other cases, the recursive definition applies. Therefore the recursive definition with its base cases, is

$$c(n,k) = \begin{cases} 1 & k = 0 \\ 1 & k = n \\ 0 & k > n \\ c(n-1,k-1) + c(n-1,k) & 0 < k < n \end{cases}$$

Once again it is easy to write a C/C++ function that computes this recursively by applying the definition:

```
int combinations (int n, int k)
{
    if ( k == 0 || k == n )
        return 1;
    else if ( k > n )
        return 0;
    else
        return combinations(n-1, k-1) + combinations(n-1, k);
}
```

The interesting question is how to show that this always terminates. In each recursive call, n is diminished. In some, k is not diminished. Therefore, eventually either $k >= n$ or $k = 0$.

In any case, this is again a very inefficient way to compute c(n,k) and it should not be used.

4.3 Binary Search Revisited

The binary search algorithm was presented in pseudo-code earlier. Now we can work out some of the programmatic details.

- First we consider an arbitrary array, not a dictionary with pages and words on pages. The algorithm is given an array of values.
- Second we remove printing from the algorithm. An algorithm should return its results to its caller, not print them on a device. *In general, functions whose purpose is not to perform I/O should not perform any I/O, as this makes them less portable, cohesive, and reduces their performance.* The return value should be either the index in the array where the item is found or an indication that it is not in the array at all. Since array indices are always non-negative, we can use -1 to indicate that the search failed.
- Third is the issue of how to find the middle of the array. The middle is the index halfway between the top index and the bottom index of the part of the array being searched. Since the part of the array being searched must vary depending on the results of comparisons, the top and bottom of the search range will be parameters of the function.

Putting this together, the algorithm becomes

```
int  binary_search( const ordered_type theArray[],
                    int bottom,
                    int top,
                    ordered_type keyword )
{
    if ( bottom > top ) {
        // the function was called with an empty range
        return -1;
    else {  // top >= bottom
        // find the middle index
        int middle = ( top + bottom ) /2;
        // compare keyword to value at the middle
        if ( keyword < theArray[middle] )
            // keyword is not in the upper half so try again in lower half
            return binary_search( theArray, bottom, middle-1, keyword );
        else if ( keyword > theArray[middle] )
            // keyword is not in the lower half so try again in upper half
```

```
                return binary_search( theArray, middle+1, top, keyword );
            else
                // keyword >= theArray[middle] && keyword <= theArray[middle],
                // so keyword == theArray[middle] and we found it
                return middle;
        }
    }
```

Notes.

1. The array parameter is declared constant in order to prevent the code from accidentally modifying it, since it is passed by reference in C++ and passed as a pointer in C.
2. The type of the array must be a type for which comparison operations are defined; the type `ordered_type` represents an arbitrary ordered type such as int, char, or double.
3. The comparisons are ordered so that first it checks if the keyword is less than the array element, then larger, and if both fail, it must be equal. This is more efficient than checking equality first. Why?

The prototype and its contract, using Doxygen-style comments, are

```
/**
 * @precondition 0 <= bottom && top < ARRAYSIZE && for every i 0 <= i <= top-1
 *               theArray[i] <= theArray[i+1]  && ARRAYSIZE is the size of the array
 * @postcondition none (the array is unchanged )
 * @param  theArray  the array to search
 * @param  bottom    low index of range to search in the array
 * @param  top       high index of range to search in the array
 * @param  keyword   value to look for (the search key)
 * @return if keyword is in the array between bottom and top, its index
 *         otherwise -1
 */
int  binary_search( const ordered_type theArray[],
                    int bottom,
                    int top,
                    ordered_type keyword ) ;
```

Exercise 1. Suppose the array contains the integers 5, 10, 16, 17, 19, 30, 32, 33, 34, 67, 68, 69, 81, 83, 87, 91, 92 and suppose the keyword is 10.

1. What is the set of array values against which the keyword will be compared?
2. How many comparison operations will be executed?
3. What if the keyword is 3? What is the sequence of array values against which it will be compared, and how many comparisons are executed?

4.4 Towers of Hanoi

The last problem we will consider is the famous *Towers of Hanoi* problem. You are given three pegs, labeled A, B, and C. Each peg can hold n disks. Initially, n disks of different sizes are arranged on peg A in such a way that above any disk are only smaller disks. The largest is therefore at the bottom of the pile and the smallest at the top. The problem is to devise an algorithm that will move all of the disks to peg B moving one disk at a time subject to the following two rules:

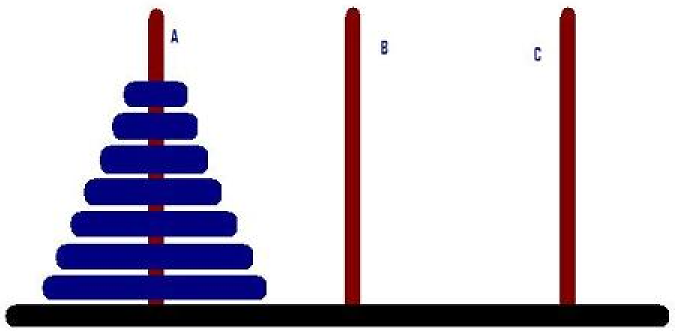

Figure 3: Towers of Hanoi

- Only the top disk can be moved from its stack; and
- A larger disk may never be on top of a smaller disk.

This is a problem with no obvious, simple, non-recursive solution, but it does have a very simple recursive solution.

1. Ignore the bottom disk and solve the problem for $n-1$ disks, moving them to peg C instead of peg B.
2. Move the bottom disk from peg A to peg B.
3. Solve the problem for moving $n-1$ disks from peg C to peg B.

Now all disks will be on peg B in the correct order.

If we write `towers(count, source, destination, spare)` to represent the algorithm that moves `count` many disks from `source` to `destination` using the `spare` as needed, subject to the rules above, then the algorithm we just described can be written as follows:

```
towers(n-1, A, C, B);
towers(1,   A, B, C);
towers(n-1, C, B, A);
```

It is astonishingly simple. The base case is when there is a single disk. Putting this together, we have

```
typedef char  peg;
void towers(int count, peg source, peg dest, peg spare)
{
    if ( count == 1 )
        cout << "move the disk from peg" << source << " to peg " <<  dest << endl;
    else {
       towers(count - 1, source, spare, dest);
       towers(1, source, dest, spare);
       towers(count - 1, spare, dest, source );
    }
}
```

This is an example of a recursive algorithm with three recursive calls. In each call the problem size is smaller. The problem size is the number of disks. The recursion stops when the problem size is 1. The function produces, as its output, a sequence of statements showing which disk was moved. Alternatively, we could just store this sequence in an array of strings that could be passed back via a parameter and then printed by the calling code.

What is not clear is how many steps this takes. If we were to measure the steps by how many times a disk is moved from one peg to another, then the interesting question is how many moves this makes when given an initial set of N disks on a peg. We will return to this when we discuss recurrence relations a bit later.

Exercise 2. Write up this algorithm and run it to verify that it works. Add statements to see how many recursive calls were made. Then, add the pre- and post-conditions to make the contract for the function's prototype.