Maggie Johnson
CS109

Recursion

Key topics:

* Recursive Definitions
* Recursive Subprograms
* Proving Properties of Recursive Programs

Recursive Definitions

One of the most important tasks in computer science is to discover and characterize regular patterns, such as those associated with processes that are repeated. In math, such patterns are called *sequences.* There are three ways to define a sequence:

1) enumerate the items in the sequence and hope the pattern is obvious;
2) give a formula for the elements in the sequence, for example, the sequence of powers of 2 is given by $a(n) = 2^n$;
3) give a *recursive* or *inductive* definition with the following parts:

   a) the initial condition or basis which defines the first (or first few) elements of the sequence;
   b) an inductive step in which later terms in the sequence are defined in terms of earlier terms. This is called a *recurrence relation.*

Here are lots of examples of these types of definitions (mostly recurrence relations):

1) We'll start with the obvious: The set of even numbers can be defined as:

1) 2, 4, 6, 8, 10….
2) 2k for some integer k
3) basis: 2 is in EVEN
   induction: if x is in EVEN then so is x + 2

2) Compound interest can be defined recursively as:

basis: A(0) = initial amount
induction: A(k) = interest rate * A(k-1)

For example, if the initial amount is $1000 and the interest rate is 1.055, after 21 years we get:

A(0) = 1000
A(1) = 1.055 * A(0) = 1055.00
A(2) = 1.055 * A(1) = 1113.02
…
A(20) = 1.055 * A(19) = 2917.76
A(21) = 1.055 * A(20) = 3078.23

3) The product of the positive integers from 1 to n, inclusive, called "n factorial" is denoted by n!: multiply 1 * 2 * ... * n = n! (this is an iterative definition). To devise a recursive definition, notice:
1! = 1 2! = 1 * 2 = 2 3! = 1 * 2 * 3 = 6 4! = 1 * 2 * 3 * 4 = 24
Observe that 4! = 4 * 3! or in general: n! = n * (n - 1)!. Factorial is defined recursively as follows:

basis: 1! = 1
induction: n! = n * (n - 1)!

For example: 4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1
2! = 2 * 1
3! = 3 * 2
4! = 4 * 6

By the way, inductive proofs come in very handy when we are working with recursive definitions. For example, we can prove using induction that the recursive definition of factorial is equivalent to the iterative one:

Prove P(n): n! as defined recursively equals 1 * 2 * ... * n

i) base case: prove that P(1) is true: 1!=1 and 1 * 1 = 1.

ii) induction hypothesis is to assume P(k): n! as defined recursively equals 1 * 2 * ... * n and show P(k+1): (n + 1)! as defined recursively equals 1 * 2 * ... * n * (n + 1).

PROOF:

(n + 1)! =	(n + 1) * ((n + 1) - 1)!	substitution of (n+1) in recursive definition
	(n + 1) * n!	subtraction
	n! * (n + 1)	commutative law for *
	(1 * 2 * ... * n) * (n + 1)	inductive hypothesis

P(k+1) is true when P(k) is true, and therefore the recursive and iterative definitions are equivalent.

4) Give a recursive definition of a^n where a is a nonzero integer and n is a nonnegative integer.

basis: $a^0 = 1$
induction: $a^{(n+1)} = a^n * a$

5) The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... i.e., each element is the sum of the two preceding elements is defined recursively:

basis: F(0) = 0, F(1) = 1
induction: F(n) = F(n-1) + F(n-2)

Induction also comes in handy when we need to prove properties of sequences and patterns. We saw this in the Induction handout. Here's an example: Using strong induction, we will prove that in the Fibonacci sequence, F(n+4) = 3F(n+2) - F(n) for all n

>= 1. Not something you may have observed in the Fibonacci series, but true nonetheless...

Prove P(n): in the Fibonacci sequence, F(n+4) = 3F(n+2) - F(n) for all n >= 1

i) base case: prove that P(1) and P(2) are true:

F(1+4) = 3F(1+2) - F(1) = 3*2 - 1 = 5 F(5) = 5
F(2+4) = 3F(2+2) - F(2) = 3*3 - 1 = 8 F(6) = 8

ii) induction hypothesis is to assume P(k): Assume that for all positive numbers i where i <= k: F(i+4) = 3F(i+2) - F(i) and show P(k+1): F(k+1+4) = 3F(k+1+2) - F(k+1) or F(k+5) = 3F(k+3) - F(k+1)

PROOF:

We know from the definition of the Fibonacci sequence that F(k+5) = F(k+3) + F(k+4). By the inductive hypothesis with i = k -1 and i = k respectively,

F(k+3) = 3F(k+1) - F(k-1)
and
F(k+4) = 3F(k+2) - F(k)

Therefore, F(k+5) = 3F(k+1) - F(k-1) + 3F(k+2) - F(k)
3[F(k+1) + F(k+2)] - [F(k-1) + F(k)]
3F(k+3) - F(k+1) substitution of inductive part of recursive definition

P(k+1) is true when P(k) is true, and therefore P(n) is true for all integers >= 1.

6) Arithmetic expressions can be defined recursively as follows:

basis: The following entities are arithmetic expressions:

- variables
- integers
- real numbers

induction: if E1 and E2 are arithmetic expressions, then the following are also arithmetic expressions:

1. (E1 + E2)
2. (E1 - E2)
3. (E1 * E2)
4. (E1 / E2)
5. if E is an arithmetic expression, so is (-E).

Show that (y * (-(x + 10))) is an arithmetic expression:

i)	x	basis
ii)	10	basis
iii)	(x + 10)	induction rule 1
iv)	(-(x + 10))	induction rule 5
v)	y	basis
vi)	(y * (-(x + 10)))	induction rule 3

Feeling frustrated with how long it takes to get email over his local network, Fred decides to fight back. He and a group of his friends decide to bring the system to its knees on a specified date in the near future. The way they are going to do this is Fred will send an email to 10 of his friends. In this email, he will ask each of his 10 friends to pass the same email on to 10 of their friends (all subscribers to the same network), etc., etc. So how long do you suppose it will take to flood the system, assuming everyone plays their part perfectly?

Recursive Subprograms

So much for the mathematical side of recursion. As computer scientists, we are most interested in how recursion can be used in programming. As you know, recursion in programming has a very basic form. As a programming technique, recursion is when a procedure or function calls itself either directly or indirectly. To keep such a procedure or function from calling itself indefinitely, a recursive subprogram must have two properties:

1) There must be some base condition for which the subprogram does not call itself.
2) Each time the subprogram calls itself, it must converge on the base condition.

Basically, recursion is a technique for performing a task T by performing another task T'. T' has exactly the same nature as the original T (we are calling the same procedure or function), but it is in some sense smaller than T. Recursion is another way of doing iteration in a program. In fact, any iterative code can be rewritten as recursive code. Sometimes recursion is more elegant and efficient than using loops, sometimes not, so most programmers use a mix of both techniques.

Consider the following silly function called Sum which returns the sum of the first n integers. This recursive function takes advantage of the fact that adding 4+3+2+1 is the same as adding 4 and the sum of 3+2+1:

```
int Sum(int n)
/* using recursion */
{
        if (n == 1)
                return(1);
        else
                return(n + sum(n - 1));
}
```

Calling this function with n = 4 gives the following execution:

```
Sum(4);
        Sum(3);
                Sum(2);
                        Sum(1);
                        sumN = 1;
                sumN = 2 + 1;
        sumN = 3 + 3;
sumN = 4 + 6                    { return 10 }
```

No computer scientist in his or her right mind would ever implement sum in this manner due to the computational and memory overhead of making recursive calls. Not to mention the fact that we have a Gauss' neat little formula that can give us the sum directly (n(n+1)/2).

Recursively define the set of bit strings that have more 0's than 1's.

Give a recursive algorithm for finding the reversal of a bit string:

Binary Search

Recall that a binary search (the "telephone directory" search) consists of comparing the item to be found with the element in the middle of the list. If the search item is less than the item in the middle, we limit our search to the first half of the list. If the search item is greater than the item in the middle, we limit our search to the second half of the list. Otherwise, the item is equal to the middle element and the item is found. If the item is not found, we do the exact same process again with a new middle element, this one being in the middle of the first or last half of the original list. We continue doing this until the item is found. Notice that the size of the list that we are searching is cut in half each time. Therefore, we are doing exactly the same type of search but the size of the input has gotten much smaller. This is an example of what we in the computer science racket call "divide and conquer!".

```
int binsearch(int x, int v[], int low, int high)
/* recursive binary search:  find x in v[low]..v[high]; return index of location  */
{
        int mid, loc;

        mid = (low + high) / 2;
        if (x == v[mid])
                return(mid);
        else if ((x < v[mid]) && (low < mid))
                return(binsearch(x, v, low, mid-1));
        else if ((x > v[mid]) && (high >  mid))
                return(binsearch(x, v, mid+1, high));
        else
                return(-1);
}
```

What does the following function output? _____

```
int XX(int n, int a)
{
     if (n == 1)
          return(a * a);
     else
          return((XX(n-1, a)) * (XX(n-1, a)));
}
```

Proving Properties of Recursive Programs

Proving the correctness of recursive programs is similar to proving the correctness of a loop. We use induction to prove the subprogram works for the base case, i.e., if the subprogram is called just once; then we assume it works for k calls, and prove that it works for k+1 calls. For example, to prove the factorial function correct:

```
int Fact(int n)
{
1)      if (n == 1)
2)              return(1);
3)      else
4)              return(n * Fact(n - 1));
}
```

Prove: When called with argument i for parameter n, Fact returns i!

i) base case: if the condition "n == 1" is true (i = 1), then the basis (line 2) is executed and Fact is assigned the value of 1. Therefore Fact(1) returns the value of 1.

ii) Assume the induction hypothesis: We assume that when called with an argument i > 1, Fact returns i!. We must show that Fact(i+1) returns (i+1)!.

PROOF:
If i > 1 then i+1 is at least 2, so the inductive part of the definition (line 4) is executed. Now, n = i+1 so the returned value is:

(i+1) * Fact(i)	
(i+1) * i!	inductive hypothesis substitution
(i+1)!	recursive definition of factorial

The k+1 recursive call has been verified when the k recursive call was correct, and therefore the recursive function is correct.

The binary search function can also be proven correct using induction. We will do strong induction on the length of array v. What we want to prove:

The function binsearch returns the index location of item x in array v which has bounds of low and high; or, it returns -1 if item x is not in the array.

```
int binsearch(int x, int v[], int low, int high)
/* recursive binary search:  find x in v[low]..v[high]; return index of location  */
{
        int mid, loc;

1)      mid = (low + high) / 2;
2)      if (x == v[mid])
3)              return(mid);
4)      else if ((x < v[mid]) && (low < mid))
5)              return(binsearch(x, v, low, mid-1));
6)      else if ((x > v[mid]) && (high >  mid))
7)              return(binsearch(x, v, mid+1, high));
8)      else
9)              return(-1);
}
```

i) base case: The base case is when the function is called only once. There are two possibilities: either the array is empty or the array has 1 element. If we have an empty array; mid has a value of 0 and the final else test (line 9) is executed. -1 is returned because x is not found. The other base case is if the array has one element. Mid has a value of 1. If x is equal to the only item in the array, line 3 is executed & binsearch returns the index location of x, being 1. If x is not equal to the only element in the list, line 9 is executed and binsearch returns -1. Thus, binsearch works properly for the two base cases.

ii) Assume the induction hypothesis: Assume that binsearch works properly for arrays of size 0, 1, ..., n. We must show it works properly for an array of n+1 elements.

PROOF:
The original call to the function is with upperbound+1, or lowerbound-1 (we have n+1 elements). If the item x is at the middle index location, we execute the first base case. If high > mid or low < mid, the item x is not in the subarray, and we execute the other base case. Otherwise, the array is split in half and we continue the search in one half or the other. The length of this new list cannot be as large as n+1; in fact, its size is covered by the inductive hypothesis. The inductive hypothesis holds and the functions works for the smaller arrays.

Thus, since binsearch works for an array of size n+1 when we assume it works for an array of size n, binsearch works on an array of any size. Just like magic!

Bibliography

For more practice with recursive definitions and algorithms, refer to Rosen, section 3.3 and 3.4.

D. Barron, *Recursive Techniques in Programming*, New York: American Elsevier, 1968.

R. Bird, "Improving Programs by the Introduction of Recursion," *Communications of the ACM*, 20:11, (Nov., 1977), pp. 434-39.

D. Hofstadter, *Godel, Escher, Bach: An Eternal Golden Braid*, New York: Basic Books, 1979.

D. Knuth, *The Art of Computer Programming, 2nd Ed.*, Menlo Park, CA: Addision Wesley, 1981.

Z. Manna and R. Waldinger, *The Logical Basis for Programming*, Reading, MA: Addision Wesley, 1990.

E. Roberts, *Thinking Recursively*, New York: Wiley, 1986.

M. Wand, *Induction, Recursion and Programming*, New York: North-Holland, 1980.

Historical Notes

Recursion was studied as early as 1202 by Leonardo Fibonacci (c.1170-c.1240) where we find probably the earliest example of a recursively defined sequence: the Fibonacci sequence (see above) was defined in terms of the reproductive characteristics of a group of rabbits. This

sequence was studied by many later mathematicians who used it to develop methods for defining a formula for F_n involving only n. This was done by Abraham de Moivre (1667-1754) and independently by Daniel Bernoulli (1700-1782) and Jacques Binet (1786-1856). By the way, the formula is:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

This group of mathematicians along with Eduard Lucas (1842-1891) created and studied many recursively defined sequences. A formula for calculating the n^{th} term of such a sequence is called a *generating function*, and the methods these mathematicians developed can be used to find such a formula for nearly any sequence. By the way, Lucas also created the Towers of Hanoi game, a very famous example of recursion (we will study this later in the quarter).

The systematic study of recursion began in the 1920's when mathematical logicians began to treat questions of decidability and computability. All the common functions of number theory, it was shown, could be defined recursively. This was the beginning of an important area of computability: *recursive function theory*. Stephen Kleene introduced this concept in "General Recursive Functions of Natural Numbers," *American Journal of Mathematics*, 57 (1935), and wrote the standard reference on this theory in 1952 (*Introduction to Metamathematics*, Princeton, NJ: Van Nostrand).

The first language to use recursive subroutines was IPL (Newell, Shaw and Simon). The language was used for their Logic Theorist system, a program that could prove propositional logic theorems. This is considered to be one of the very first AI programs. The first language to provide an automatic mechanism for recursion was Lisp. Most imperative languages after Algol-60 allowed recursion.

Binary search was first mentioned by John Mauchly (of ENIAC and UNIVAC fame). The method became well known in the 50's, but no one seemed to have worked out the details of what should be done when N is not a power of 2. H. Bottenbruch was apparently the first to publish a binary search algorithm that works for all N in 1962.