CS 3410 – Ch 7 – Recursion

Sections	Pages
7.1-7.4, 7.7	293-319, 333-336

7.1 Introduction

1. "A *recursive method* is a method that either directly or indirectly makes a call to itself." [Weiss]. It does this by making problem smaller (simpler) at each call. It is a powerful problem solving technique in some situations, but can be tricky to model and implement. So, in this chapter we revisit the topic of recursion. Another way to think about recursion is that something recursive is defined in terms of itself. Perhaps the simplest example is calculating factorial: $n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$. However, we can also see that $n! = n \cdot (n-1)!$. Thus, factorial is defined in terms of itself.

Iterative Solution

```
static double factorial( double n )
{
   double sum = 1.0;

   for( int i=1; i<=n; i++ ) sum *= i;

   return sum;
}
```

Recursive Solution

```
static double factorial( double n )
{
   if( n <= 1 )
      return 1;
   else
      return n * factorial( n-1 );

}
```

7.2 Mathematical Induction

1. Recursion is based on the mathematical principle of induction. Assume that you have some type of mathematical statement: $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$ and wish to prove that it is true. One technique is *mathematical induction*, which is carried out in three steps:

   a. Verify that the statement (algorithm) is true for a small case (say, n=1 or n=2, *etc.*)
   b. Assume that it is true for n=k.
   c. Using the assumption, show that the statement is true for n=k+1.

2. Example – Theorem: $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$. Note that there is a recursive definition of this statement: $sum(n) = n + sum(n-1)$.

   Proof:

   a. Verify for n=1: $LHS \overset{?}{=} RHS$

   $LHS = \sum_{i=1}^{n} i = \sum_{i=1}^{1} i = 1$

   $RHS = \frac{n(n+1)}{2} = \frac{1(1+1)}{2} = 1$

b. Assume statement is true for $n = k$, $\sum_{i=1}^{n=k} i = \frac{k(k+1)}{2}$

c. Using the assumption, show that the statement is true for n=k+1.

$$\sum_{i=1}^{k+1} i \overset{?}{=} \frac{(k+1)(k+2)}{2}$$

$$LHS = \sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^{k} i = (k+1) + \frac{k(k+1)}{2} =$$

$$\frac{2(k+1)}{2} + \frac{k(k+1)}{2} = \frac{(k+1)(k+2)}{2} = RHS$$

Homework 7.1

1. Prove by induction: $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$.
2. Prove by induction: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

7.3 Basic Recursion

1. A recursive method for the problem of summing the first *n* integers:

```
public static int sum( int n )
{
      if( n==1 )
            return 1;
      else
            return sum(n-1) + n;
}
```

We frequently say that such a method, *calls itself*. Actually, it is calling a clone of itself with a different set of parameters. At any one time, only one clone is active. The rest are on the stack awaiting their return to active status.

2. Two of the four fundamental rules of recursion:

   a. Base Case: You must always have at least one situation where the solution to this smallest(er) problem is automatic. That is, there is no recursion. We often say that the base case stops the recursion.
   b. Make Progress: All recursive calls must make progress towards a base case. We often say that a recursive call must make the problem smaller.

3. Recursion requires bookkeeping to keep track of the variables and parameters. When a recursive call is make, all variables in the new invocation are placed on the stack and become active. When a base case is encountered, an instance of the method ends, the stack is popped, and the most recent, previous invocation becomes active. For deep recursive chains, the stack can get large and the computer can run out of memory. Recursive methods are (at best) slightly more time consuming than equivalent non-recursive techniques because of the bookkeeping involved. It is important to remember how variables are passed in Java. We often think (incorrectly), "primitives are passed by value and objects are passed by reference." The first part is correct, primitives are passed by value. Actually, in Java, you have a *pointer* to an object and that *pointer* is passed by value. A good reference on this: http://javadude.com/articles/passbyvalue.htm. Thus, if you reassign the pointer to a new object, the calling program will not see this:

```
public static void main( String [ ] args )
{
   Blob b = new Blob(33);

   reassign( b );

   System.out.println( b.n ); // 33

}
public static void reassign( Blob b )
{
   b = new Blob(44);
}
```

However, if you use the pointer to change the object, then the change is seen:

```
public static void main( String [ ] args )
{
   Blob b = new Blob(33);

   change( b );

   System.out.println( b.n ); // 44

}
public static void change( Blob b )
{
   b.n = 44;
}
```

4. Suppose we have a language where we don't have a way to print numbers, but we do have a way to print characters. Write an algorithm to print a number:

```
      public static void printNumber( int n )
      {
1           if( n>=10 )
2                printNumber( n/10 );
3           System.out.print( Integer.toString(n%10).charAt(0) );
      }
```

Proof by induction: Let *k* be the number of digits.

a. Verify for $k = 1$: For a one-digit number, no recursive call is made. Line 3 immediately prints out the digit as a character. The statement is true because any one-digit number, mod 10, is the one-digit number itself.

b. Assume that for any $k > 1$ the method works correctly. Thus, any *k*-digit number prints correctly.

c. Now, using the assumption, show that the method is correct for a $k + 1$ digit number.

Since $k > 1$ a recursive call is made at line 2. There, a $k + 1$ digit number is converted to a k-digit number in line 2, comprised of the first k digits (left-to-right), by the integer division, $n/10$, which prints correctly by assumption. Finally, at the conclusion of the recursive printing of the *k*-digit number (the first *k* digits), line 3, prints the $k + 1st$ digit (the last digit) because any number (n), modulus 10, gives the last digit.

5. The author presents the 3rd of 4 rules of recursion: "You gotta believe," where he says that you must assume that the recursive call works. He states that many times it is too complicated to try to trace recursive calls, so you must "trust" that the recursion works (makes progress towards a base case). However, you can't "believe" just anything you type in the computer! Really, what he is saying is that if you can prove through induction that the recursions works then it really works! In other words, as you design a recursive method, try to focus on proving that it works through induction as opposed to tracing the recursive calls. With that said, tracing the recursion almost always comes in handy.

Homework 7.2

1. Prove by induction that this method correctly gives the sum of the first *n* integers:

```
1.  public static int sum( int n )
2.  {
3.     if( n==1 ) return 1;
4.     else return sum(n-1) + n;
5.  }
```

2. A palindrome is a string that reads the same from the left and right (*e.g.* godsawiwasdog). (a) What is(are) the base case(s) for the method shown below? (b) Prove by induction that the method correctly determines if an input string is a palindrome.

```
1.  public static boolean isPalindrome( String s )
2.  {
3.     if( s.length() <= 1 ) return true;
4.     else if ( s.charAt(0) != s.charAt(s.length()-1) ) return false
5.     else return isPalindrome( s.substring(1, s.length()-1) );
6.  }
```

7.3.3 How Recursion Works

1. Java implements methods as a stack of *activation records*. An activation record contains information about the method such as values of variables, pointers to the caller (where to return), *etc*. When a method is called, an activation record for that method is pushed onto the stack and it becomes active. When the method ends, the activation stack is popped and the next activation record in the stack becomes active and its variables are restored.

2. The space overhead involved in recursion is the memory used to store the activation records. The time overhead is the pushing and popping involved in method calls. The close relation between recursion and stacks suggests that we can always implement a non-recursive solution by using an explicit stack, which presumably could be made more efficient in time and space than the one that the JVM maintains for us when we use recursion. However, the code would be longer and more complex, generally. Modern optimizing compilers have lessened the overhead of recursion to the point that it is rarely worthwhile to rewrite an algorithm without recursion.

7.3.4 Too Much Recursion can be Dangerous

1. When should you use recursion? When it leads to a natural, less complicated algorithm, but never when a simple loop could be used, nor when duplicate work results. A good example of this last point is using recursion to calculate a Fibonacci number.

2. We remember that a Fibonacci number can be recursively defined as: $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$, where $F(0) = 0, F(1) = 1$. The corresponding method:

figure 7.6

A recursive routine for Fibonacci numbers: A bad idea

```
1    // Compute the Nth Fibonacci number.
2    // Bad algorithm.
3    public static long fib( int n )
4    {
5        if( n <= 1 )
6            return n;
7        else
8            return fib( n - 1 ) + fib( n - 2 );
9    }
```

3. The problem, here is that when we compute $fib(n-1)$ in line 8 we implicitly compute $fib(n-2)$, because by definition, $fib(n-1) = fib(n-2) + fib(n-3)$. Then, when $fib(n-1)$ completes, we explicitly compute $fib(n-2)$ again (on the right-hand side). Thus, we have made 2 calls to $fib(n-2)$, not one. In this case, it gets much worse, as the recursion continues; we get more and more duplication.

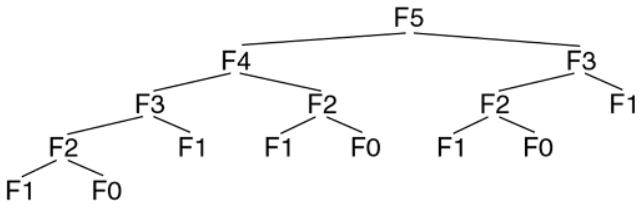


figure 7.7

A trace of the recursive calculation of the Fibonacci numbers

4. For $n \geq 3$ it can be proved that the number of calls to $fib(n)$ is larger than the actual Finonacci number itself! For instance, when $n = 40$, $F(40) = 102{,}334{,}155$ and the total number of recursive calls is more than 300,000,000. This recursive algorithm is exponential. Of course we can use a loop that iterates from say 2 to n, for a linear algorithm.

5. The author's 4th rule of recursion is the *Compound Interest Rule* which states that we should never duplicate work by solving the same instance of a problem in separate recursive calls.

Homework 7.3

1. Prove the following identities relating to the Fibonacci numbers.
   a. $F_1 + F_2 + \cdots + F_n = F_{n+2} - 1$
   b. $F_1 + F_3 + \cdots + F_{2n-1} = F_{2n}$
2. Describe the 4 rules of recursion.
3. What is an activation record and how is it used in the implementation of recursion within the JVM.
4. Explain any limitations of recursion.
5. When should recursion be used?

7.3.5 Preview of Trees

1. A *tree* is a fundamental structure in computer science. We will see uses for trees later, but for now we will introduce trees as they can be defined recursively.

2. First, a non-recursive definition of a tree is that it consists of a set of *nodes* which are connected by a set of edges.

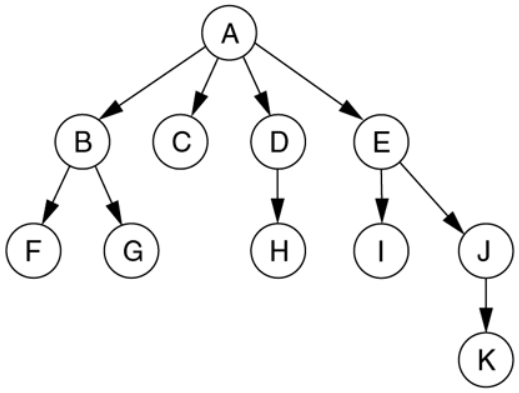


figure 7.9
A tree

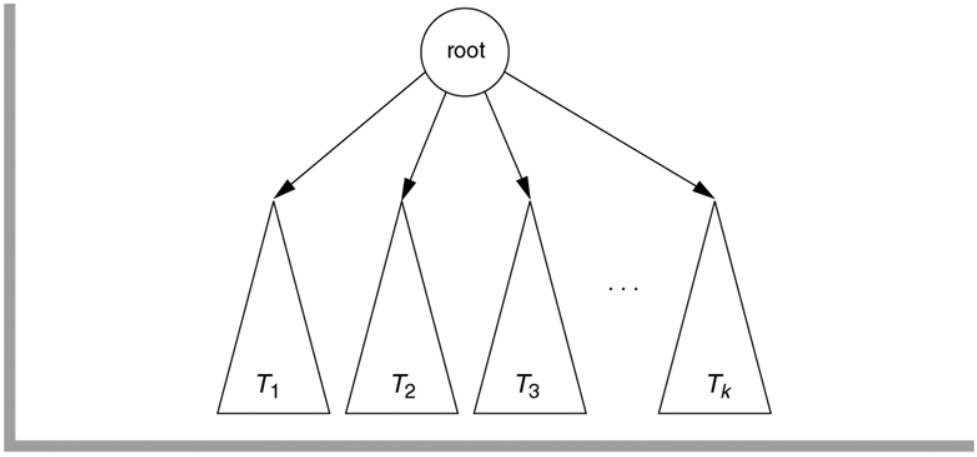
3. In this course, we will consider a *rooted tree* which has the following properties:

   a. One node is distinguished as the *root*.
   b. Every node, c, except the root, is connected by an *edge* from exactly one other node, p. Node p is $c's$ *parent* and c is one of $p's$ *children.*
   c. A unique path traverses from the root node to each node. The number of edges that must be followed to that node is the *path length*.

   A node that has no children is called a *leaf*.

4. A recursive definition of a tree is: either a tree is empty or it consists of a root node with zero or more subtrees, each of whose roots are connected by an edge from the root.

figure 7.8
A tree viewed recursively



7.3.6 Additional Examples

1. Factorial – (a) What is the complexity? (b) Prove that the algorithm works via induction.

```
1     // Evaluate n!
2     public static long factorial( int n )
3     {
4         if( n <= 1 )    // base case
5             return 1;
6         else
7             return n * factorial( n - 1 );
8     }
```

figure 7.10

Recursive implementation of the `factorial` method

2. Binary Search – (a) What is the complexity? (b) Prove that the algorithm works via induction.

figure 7.11

A binary search routine, using recursion

```
1     /**
2      * Performs the standard binary search using two comparisons
3      * per level. This is a driver that calls the recursive method.
4      * @return index where item is found or NOT_FOUND if not found.
5      */
6     public static <AnyType extends Comparable<? super AnyType>>
7     int binarySearch( AnyType [ ] a, AnyType x )
8     {
9         return binarySearch( a, x, 0, a.length -1 );
10    }
11
12    /**
13     * Hidden recursive routine.
14     */
15    private static <AnyType extends Comparable<? super AnyType>>
16    int binarySearch( AnyType [ ] a, AnyType x, int low, int high )
17    {
18        if( low > high )
19            return NOT_FOUND;
20
21        int mid = ( low + high ) / 2;
22
23        if( a[ mid ].compareTo( x ) < 0 )
24            return binarySearch( a, x, mid + 1, high );
25        else if( a[ mid ].compareTo( x ) > 0 )
26            return binarySearch( a, x, low, mid - 1 );
27        else
28            return mid;
29    }
```

7.4 Numerical Applications

These examples are useful in data security including encryption.

7.4.2 Modular Exponentiation

Consider how we would compute: $x^n \bmod p$, efficiently. An obvious linear algorithm simply multiples x by itself n times, taking the mod each time to control the size of the numbers. A faster algorithm is based on this observation:

if n is even

$$x^n = (x \cdot x)^{n/2}$$

else if n is odd

$$x^n = x \cdot x^{n-1} = x \cdot (x \cdot x)^{\lfloor n/2 \rfloor}$$

```
1   /**
2    * Return x^n (mod p)
3    * Assumes x, n >= 0, p > 0, x < p, 0^0 = 1
4    * Overflow may occur if p > 31 bits.
5    */
6   public static long power( long x, long n, long p )
7   {
8       if( n == 0 )
9           return 1;
10
11      long tmp = power( ( x * x ) % p, n / 2, p );
12
13      if( n % 2 != 0 )
14          tmp = ( tmp * x ) % p;
15
16      return tmp;
17  }
```

figure 7.16

Modular exponentiation routine

What is the complexity of this algorithm?

7.4.3 GCD

The greatest common divisor (gcd) of two nonnegative values, A and B is the largest integer D that divides both A and B. A fact that is useful for efficiently computing the *gcd* is the recursive relationship: $\gcd(A, B) = \gcd(B, A \bmod B)$. For the base case, we use the fact that: $\gcd(A, 0) = A$. For example:

$\gcd(70,25) = \gcd(25,70 \bmod 25) = \gcd(25,20)$

$\gcd(25,20) = \gcd(20,25 \bmod 20) = \gcd(20,5)$

$\gcd(20,5) = \gcd(5,20 \bmod 5) = \gcd(5,0)$

$\gcd(5,0) = 5$

figure 7.17
Computation of greatest common divisor

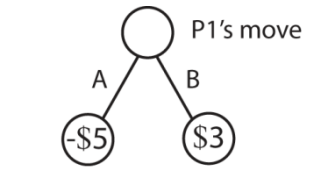
```
1     /**
2      * Return the greatest common divisor.
3      */
4     public static long gcd( long a, long b )
5     {
6         if( b == 0 )
7             return a;
8         else
9             return gcd( b, a % b );
10    }
```

It can be shown that this algorithm has complexity $O(log\ n)$. The reason is that in two recursive calls the problem is reduced at least by half.

7.7 Backtracking

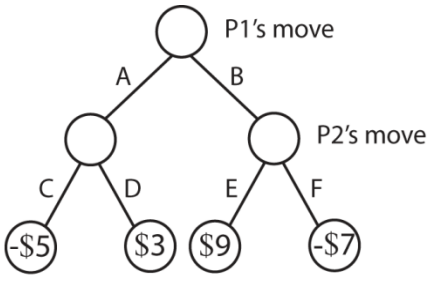
Sometimes we need algorithms that can make decisions or choices among many alternatives which depend on one another. For instance, we may want a program that can find its way through a maze. At any point in time, there are different choices about the direction to follow and a decision to follow one leads to more choices. Backtracking is a technique where we use recursion to try all the possibilities. The basis for making a decision is called a strategy.

Suppose that Player 1 (p1) and Player 2 (p2) are playing a game and that at a certain point in a game, p1 has two choices: A, with a payoff of $10 or B, with a payoff of $5. Which strategy should p1 choose?



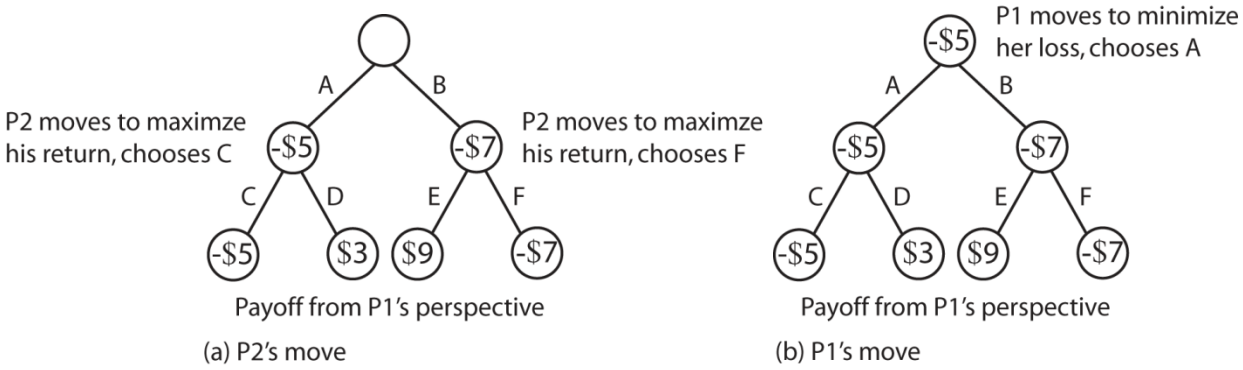
Payoff from P1's perspective

Now, suppose that p2 will get another turn after p1 as shown in the figure.. If p1 chooses A, then p2 can choose either C or D with the payoffs shown. Similarly, if p1 chooses B, then p2 can choose either E or F with the payoffs shown. All payoffs are from the perspective of p1, positive values are good for p1 and negative values are good for p2. For instance, a payoff of -$5 means that p1 pays p2 $5 and a payoff of $5 means that p2 pays p1 $5.

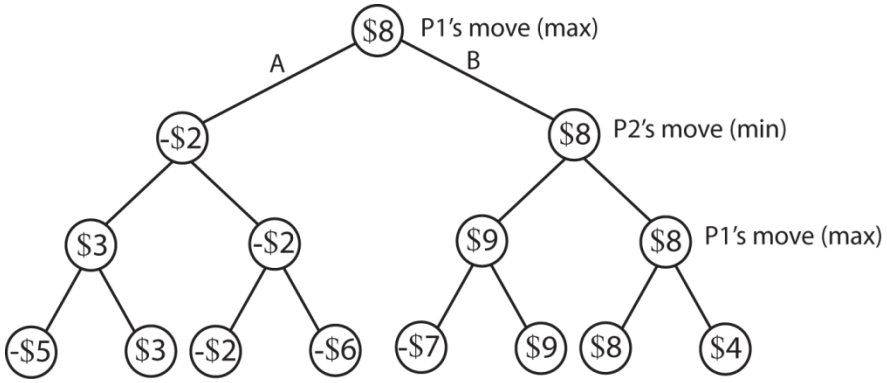


Payoff from P1's perspective

The minimax strategy seeks to minimize the opponent's maximum payoff and is based on optimal play by both players. Each leaf node is designed to have a value which can be determined by an evaluation function. The minimax strategy works backwards through the tree to aggregate node values. Each time it moves up the tree, each node chooses either the maximum or minimum of its children's nodes. This corresponds to p1 making his choice by choosing the node that maximizes his payoff while p2 chooses nodes that maximizes her payoffs.



This process of minimizing and maximizing continues to a desired depth or leaf nodes are encountered. In the example below, P1 chooses B.



A basic algorithm for finding the minimax value for a node is:

```
int minimax( node, player )

      if node is a leaf
            return value of node
      else
            for each child node

                  v = minimax( child, otherPlayer )

                  if( player == 1 )
                        if( v > best )
                              best = v
                  else if( player == 2 )
                        if( v < best )
                              best = v
            return best
```

Note that this does not tell us which choice to make, it simply returns the best value. In the context of a real game, we will have to modify this so that we explicitly keep track of the decisions/choices (*e.g.* A, B, *etc*.).

Consider the game of tic-tac-toe where a real person (p1, uses X) is playing against the computer (p2, uses O). We model the situation such that we want the computer (p2) to win. In this scenario, p1 makes a mark on the board. Then p2 (computer) must decide which move to make. If there is a choice that will end the game with p2 winning, then p2 should choose this (obviously!). If none of the choices lead to an immediate p2 win, then we can apply the minimax strategy to decide which choice p2 should make. To do so, we will think of a tic-tac-toe board as a node. The state of the board is the positioning of X's and O's. Since we are modeling from the perspective of the computer, we will evaluate a board where O's win with a value of 1, where X's win with a value of -1, and 0 for a tie.

Homework 7.4

1. Describe in detail the minimax strategy.
2. Describe how the minimax strategy is used in the game of tic-tac-toe.
3. Describe in detail how backtracking is used in HW 2.