Module 2

Recursion

Objectives:

1. Understand what recursion is and how it can be useful.
2. Be able to identify the bases case(s) and the recursive cases(s) for a recursive problem.
3. Be able to estimate running time for a recursive method.
4. Be able to recognize cases where recursion can be very inefficient.
5. Understand the concept of dynamic programming.
6. Understand the notion of backtracking and how it can be used.
7. Be able to use the box trace method to trace the execution of a recursive method.
8. Be able to apply recursion to solve a programming problem.

Textbook:

Chapter 9, sections 9.1 - 9.31.

Chapter 14, sections 14.1 - 14.33.

Labs:

1. Do the Java-based recursion demo lab.
2. Do the box trace lab (this is a Flash animation).

Assignment:

Complete the homework for Module 2.

Quiz:

Complete the quiz for Module 2.

Intro:

Recursion, very simply put, is when a method body contains a call to the method itself. Recursion can be used to write relatively short, very elegant code to solve certain types of very complex problems - namely, problems that can be broken down repeatedly into smaller and smaller versions of the same problem. Recursion can be tricky to understand, and it can be easily abused, but it does come in very handy at times.

Relation to Mathematical Induction:

Recall from discrete math that a recursive sequence can be defined as a combination of one or more base cases along with a recursive case, which can be defined using a *recurrence relation*. Any problem that can be expressed recursively using mathematics has a recursive solution. Mathematical induction is the basis for proving the validity of a recurrence relation, and can be used to prove that a recursive solution will work.

For any problem that can be broken down into smaller, identical problems, if it can be shown that:

1. a given solution to the problem works for all problems of size 1 <= N <= *k*, and

2. the same solution will also solve a problem of size N = *k* + 1

then the solution will also work for all problem sizes. Note that it is not necessary to prove that the solution works for all values of N. You only need 3 cases:

1. a base case, which is the smallest possible version of the problem and can be solved without using recursion

2. some case of size k, which is larger than the base case, and can be solved using recursion

3. a case of size *k* + 1

Fundamental Rules of Recursion

1. There must be a base case that can be solved without using recursion; without a base case recursion will never terminate.

2. Every recursive call must progress towards the base case; i.e., every recursive call must operate on a smaller version of the problem. If this does not happen, the base case will never be reached, and the recursion will never terminate.

3. Always assume the recursive calls work. This assumption is necessary since it is impractical (and often impossible) to prove the solution will work for every single case. Of course, it is assumed you can show the algorithm works for the base case and at least one recursive case.

4. If you have multiple recursive calls in an algorithm, make sure they don’t solve the same instance of a problem more than once, or the algorithm will perform very inefficiently. The degree of inefficiency will rise exponentially as the problem size increases.

5. Any problem that can be solved using recursion can also be solved using iteration (i.e., a loop). (The reverse statement—that any problem that can be solved using iteration can also be solved using recursion—also holds true, although it does **NOT** hold true that the recursive solution will be more efficient than the iterative one).

Direct vs. Indirect Recursion

In direct recursion a recursive method calls itself, as shown in Figure 2.1.

```
public int foo(int n)
{
    if (n == 0)
        return 1;
    else
        return n + foo(n - 1);
}
```

Figure 2.1. An example of direct recursion.

In indirect recursion a recursive method calls itself via another method, as shown in Figure 2.2. Here the method foo never calls itself directly, but it can call itself indirectly via the method bar. However, this will happen only when n > 50. Indirect recursion can be very difficult to trace. What I've shown here is a relatively simple example.

```
public int foo(int n)
{
    if (n == 0)
        return 1;
    else
        return n + bar(n - 1);
}

public int bar(int x)
{
    if (x > 50)
        return foo(x);
    else
        return x % 10;
}
```

Figure 2.2. An example of indirect recursion.

Problems with Recursion

Recursion can be a great way to write a solution to a very complicated problem using a very small amount of code, but it isn't always guaranteed to outperform an iterative solution.

The biggest problem with using recursion is that it is not very efficient in terms of resources. Every call to a recursive method places what is known as an *activation record* on the stack the computer uses to keep track of the currently active method. An activation record contains copies of a method's parameters and local variables, among other things. If too many recursive calls must be made before the base case is reached, the stack will overflow and the program will crash from a stack overflow error.

Once the base is reached, the result of each recursive call is passed down the stack to previous method call. This process continues until the original recursive call, which started the recursion, is reached. If a second recursive call is made before the original recursive call is reached, a new series of activation records will be placed on the stack. The Fibonacci example, shown in Figure 2.3, illustrates how this can happen.

```
public static long fib(long n)
{
    if (n <= 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Figure 2.3. A recursive algorithm for computing Fibonacci numbers.

The recursive case in this example contains two recursive calls, `fib(n - 1)` and `fib(n - 2)`. These recursive calls are processed from left to right, so the call to `fib(n - 2)` cannot be made until the recursion has been completed for the call to `fib(n - 1)`.

The Fibonacci example also illustrates how recursion can result in the same instance of a problem being solved more than once, which has a dramatic effect on the performance of this algorithm.

Divide and Conquer Algorithms

Divide and Conquer is a very old technique commonly used to solve problems that are too large or too complex to solve as a single problem. To simplify matters, the problem is broken down, or *divided* into a set of smaller problems. Each smaller problem is then solved, or *conquered*.

Not all divide and conquer algorithms are recursive, and vice versa. Generally a recursive algorithm is not considered a divide and conquer algorithm unless it contains at least 2 recursive calls, since division of any problem must yield at least 2 separate subproblems.

The Maximum Contiguous Subsequence Sum Problem, Revisited

Recall from Module 2 that with some optimization we were able to obtain a linear algorithm for the maximum contiguous subsequence problem. A divide and conquer approach can also be used to divide the array into 2 half-sized arrays, and recursively process each half of the array. In this case the maximum contiguous subsequence can occur in one of the three following ways:

1. it is entirely in the left half
2. it is entirely in the right half
3. it spans both halves, beginning in the left half and ending in the right half

The third case is the worst, since it requires the entire array to be examined. Scanning the entire array is a linear process, and thus requires O(N) time. The process of repeatedly halving the array requires O(log N) time, since you can only repeatedly halve the array log N times (remember we are dealing with base 2 logarithms here). The linear work needed to scan the array is required in addition to the time required for the repeated halving, therefore the total running time for the algorithm will be O(N log N).

In general, for algorithms that use repeated halving:

1. Repeatedly dividing a problem into 2 half-sized subproblems requires O(log N) time, since you can only repeatedly halve anything log N times (remember we are dealing with base 2 logarithms here). So the minimum time required for a divide and conquer algorithm that always divides a problem into 2 equal sized halves is at least O(log N).

2. Dividing a problem into equal sized parts is crucial — if the parts are not equal sized, the running time can be longer than O(log N).

3. If additional work is required at each level of division, the total running time will be the product of O(log N) and the time required for the additional work.

Dynamic Programming

Recall how inefficiently the recursive Fibonacci algorithm performed, since it calculated some Fibonacci numbers more than once. It is possible to improve the performance of the recursive solution while keeping the elegance of the code. The key is to make sure each Fibonacci number is only calculated once, which can be done by keeping track of which Fibonacci numbers have already been calculated. This is also known as *dynamic programming*. An example of a recursive Fibonacci algorithm that uses dynamic programming is shown in Figure 2.4.

```
public long dynamicFibRecursive(int n)
{
    if (knownFib[n] != 0)
        return knownFib[n];

    long sum = n;

    if (n < 0)
        return 0;

    if (n > 1)
        sum = dynamicFibRecursive(n - 1) + dynamicFibRecursive(n - 2);

    knownFib[n] = sum;

    return knownFib[n];
}
```

Figure 2.4. A Fibonacci algorithm that uses dynamic programming to keep track of previously computed values.

This algorithm uses the array `knownFib` to keep track of the Fibonacci numbers that have already been calculated. For a given value of n, `knownFib[n]` will be 0 if that Fibonacci number has not yet been calculated; otherwise it will contain the Fibonacci number for n. This algorithm actually has 3 base cases:

1. if the Fibonacci number for n has already been calculated, simply return the value stored in `knownFib[n]`

2. if n < 0, return 0

3. if n == 1 and the first Fibonacci number has not been calculated yet (in this case, execution falls through to the statement `knownFib[n] = sum;`); this case is not very easy to spot at first glance

The recursive case applies if n > 1 and the nth Fibonacci number has not previously been calculated.

This algorithm is much more efficient than the original recursive algorithm.

Tracing Recursive Algorithms

Tracing the execution of a recursive algorithm can be tricky, because it is necessary to keep track of the entire sequence of recursive calls. Since each recursive call is a copy of the same method call, only with different values for one or more variables, the values of all copies of all variables must also be maintained. The easiest way I have found to trace a recursive algorithm is to use a modified form of the **box trace method**. This can be done on paper, or in digital form using any of a number of different programs (word processors, spreadsheets, drawing programs, etc.). Using the box trace method, each copy of a recursive method call is represented as a rectangle, or box. Inside the box, the values for that particular copy of the method call are stored. Subsequent recursive method calls are usually shown either to the right of, or above, the prior recursive method calls. One-way arrows from prior method calls to subsequent method calls are used to show the execution path as the recursion proceeds. Typically, there will be one or more variables whose values depend on what gets returned from the next method call, so placeholder symbols are used for these. When the base case is reached, execution proceeds in the reverse order (i.e., subsequent method calls towards prior method calls), so another set of one-way arrows going back towards the prior method calls are drawn. As the recursion proceeds back towards the initial method call, the placeholder symbols in prior method calls are replaced by the return values of the subsequent method calls. When the initial method call is reached, the recursion is complete, and the final result should be obtained.

To better illustrate how this technique works, there are a couple of labs that accompany this module. One lab is a Flash animation that runs through execution of a recursive algorithm to calculate factorial numbers. The other lab is written in Java, and illustrates how the recursive Fibonacci algorithm works. The Flash animation uses the left to right progression for the boxes, while the Java lab places the boxes vertically, like a stack.

Preview of Trees

We will be spending a lot of time this semester talking about different types of trees and how they are used. This chapter gives a preview of how to look at trees recursively.

A tree consists of a collection of *nodes* that store data items. Each *parent node* can have zero or more *child nodes*. The node at the top of the tree is the root node, and is the parent node for the entire tree. Parent nodes are connected to their child nodes via unidirectional edges. A parent node will contain references to each of its child nodes, but no child node will have a reference to its parent node.

A tree can be defined recursively since each parent node can be considered the root node of a subtree. The base case would be a parent node that has zero child nodes. Such a node is referred to as a leaf. The recursive case would apply to any parent node that has at least one child node. In the example tree shown in Figure 2.5, the root node is node A. The root node has 2 subtrees, with nodes B and C being the roots of those subtrees. Nodes E, F, G, H and I are all base cases since they have no child nodes.

The ability to define trees recursively becomes important when it is necessary to iterate over all the nodes in a tree. If one begins at the root node, it is easy to iterate through all the nodes of one individual path in a tree, but once a leaf node is reached, how does one reach the nodes that are not on that particular path? Although a loop can be written to do this, recursion will make the task much easier, as we will see in a few weeks.

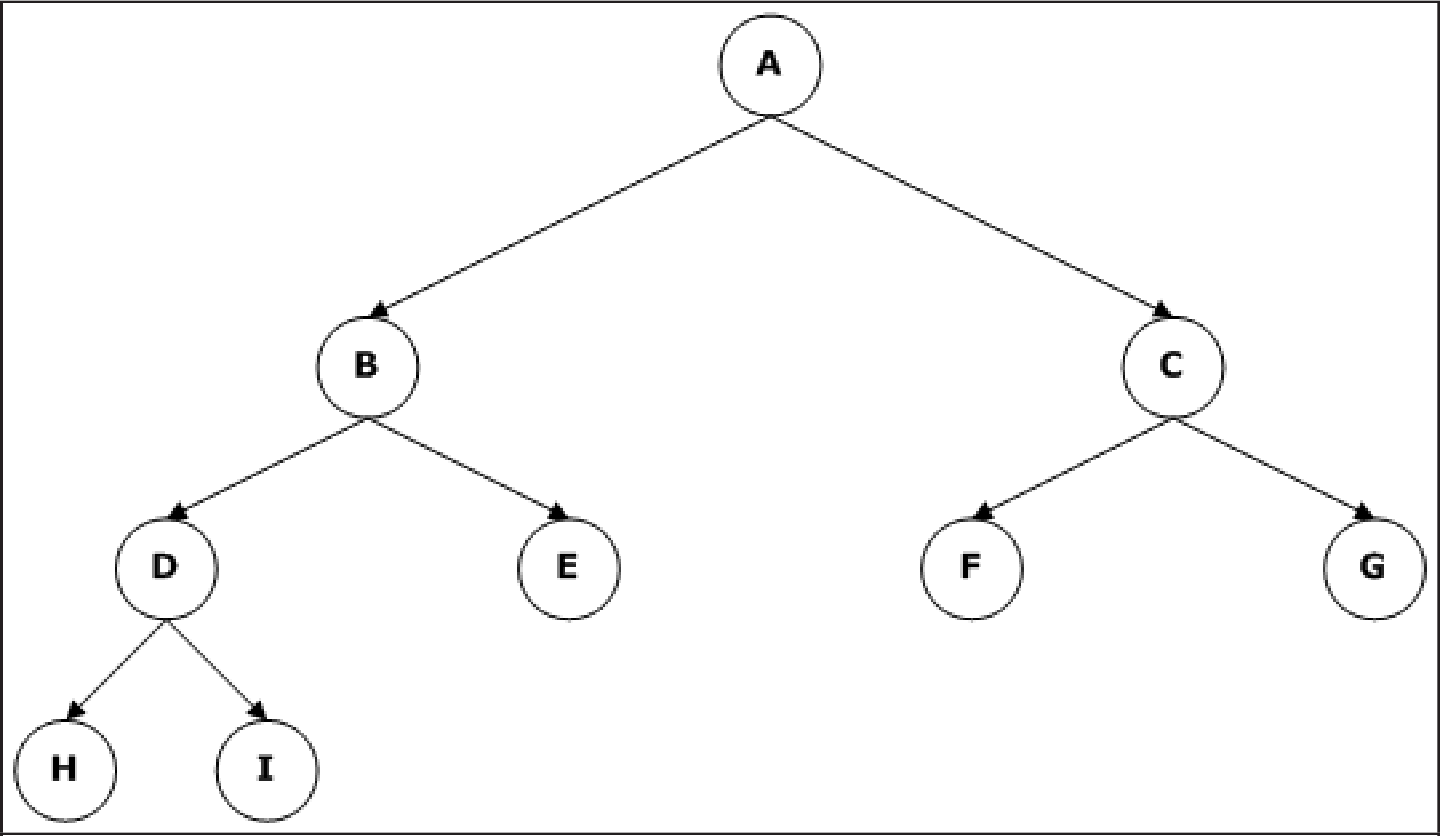

Figure 2.5. A generic tree.

Backtracking

Recursion is often useful in brute-force algorithms, where all combinations for a given problem must be exhaustively searched. Examples of such problems are:

1. finding the best move for a game such as Tic-Tac-Toe, given a particular game state
2. finding a path through a maze
3. the classical Eight Queens problem, where the goal is to place 8 queens on a chessboard such that none of the queens can be attacked

When recursion is used in this way it is known as **backtracking**, since after an algorithm explores one possible path it backtracks along that path until it finds a path it has not yet examined. Eventually all possible paths will be examined.

<u>Backtracking example - finding the best move in a Tic Tac Toe game</u>:

Given a blank Tic Tac Toe board, the first player has 9 squares to choose from. The second player will have 8 squares to choose from, once the first player has made a move. On the first player's second move there will be 7 empty squares left, and so on. Some moves will lead to a winning game, while other moves will lead to a loss or a draw. In order to determine the best move, given a particular state of the game board, it is necessary to exhaustively examine all possible moves and countermoves. The entire set of possible moves can be represented as a tree. A partial tree is shown below in Figure 2.6. Each node contains a particular board state; i.e., the combination of X's and O's that have been placed on the board so far. Finding the best move involves traversing every possible path down the tree from a given node, or board state. Note that I have only expanded one node for each level. In the complete tree all the nodes at each level would be expanded as shown.

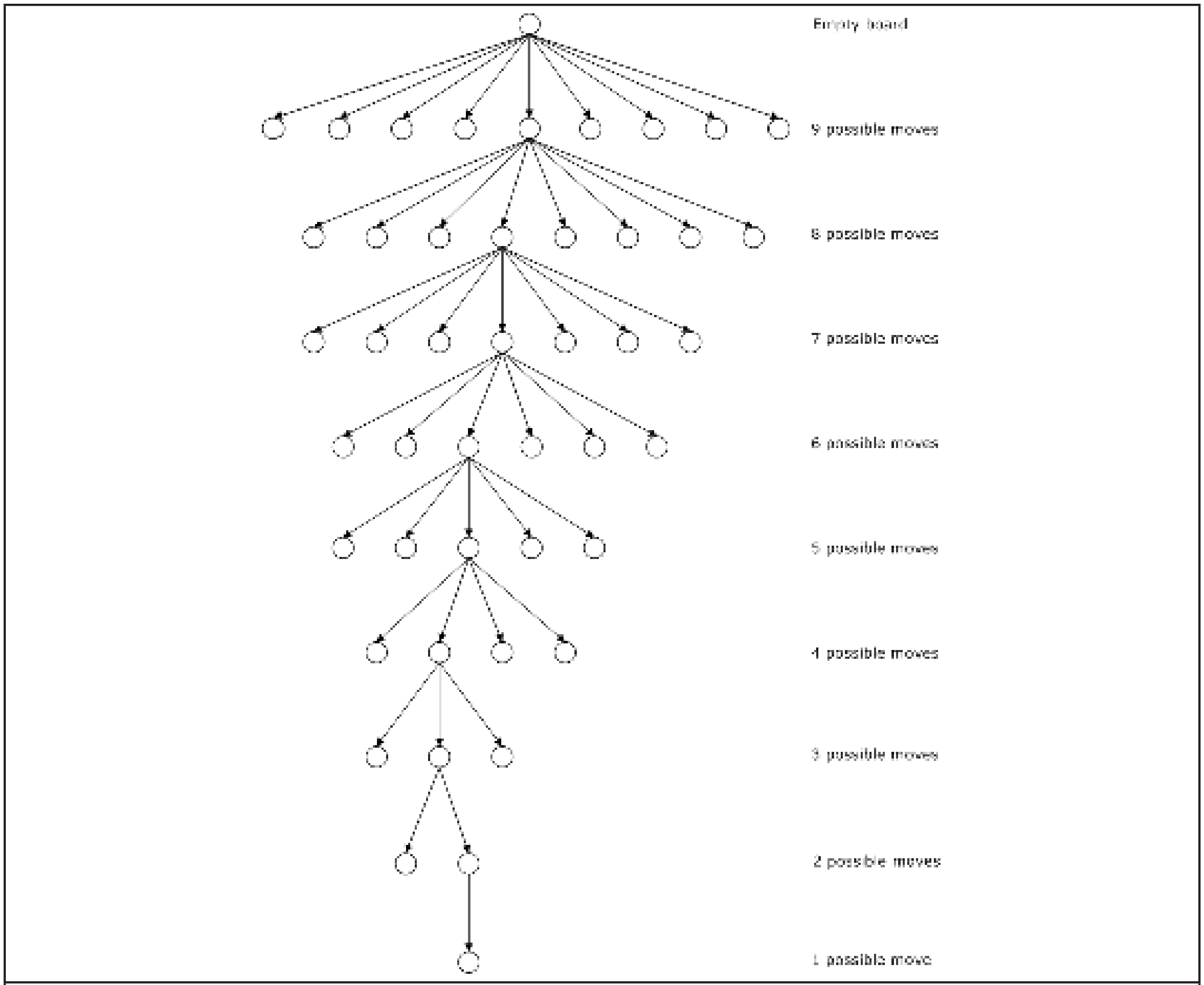

Figure 2.6. A partial tree for the game of Tic Tac Toe.

Recursion vs. Iteration

Keep in mind that any problem that can be solved recursively can also be solved iteratively (i.e., using a loop). Which method is better? In general, for most problems an iterative solution will usually be more efficient than a recursive solution. Recursion is intrinsically less efficient, not necessarily in terms of running time, but because each recursive call consumes stack space in memory. If too many recursive calls are made, the stack can overflow, bringing your program to a screeching halt. Recursion is also somewhat less efficient in terms of running time, since once the recursion has been set into motion it must continue until a base case has been reached. Once the base case has been reached, execution must proceed down the stack of activation records until the statement that initiated the recursion has been reached.

This doesn’t mean recursion is bad practice, or that it should not be used. Solutions to very complex problems can be extraordinarily difficult to write using iterative code. In these cases recursion is a good choice. You just need to be aware that for very large problems the amount of available memory may turn out to be a limiting factor. However, if an iterative solution can be easily written, use it instead.