Chapter 6: Recursion

1. Triangular Numbers
2. Factorials
3. Anagrams
4. A Recursive Binary Search
5. The Towers of Hanoi
6. Mergesort
7. Eliminating Recursion
8. Some Interesting Recursive Applications

---

Introduction

• A programming technique in which a function calls itself.
• One of the most effective techniques in programming.
Triangular Numbers

- Consider the numbers 1, 3, 6, 10, 15….  
- What is so peculiar about them?  
- The \( n^{th} \) term in the series is obtained by adding \( n \) to the previous number.  
- Looping can be used to find the \( n^{th} \) term.  
- Recursion can also be used to find the \( n^{th} \) term.

![Figure 6.1: The triangular numbers](image)

Finding \( n^{th} \) Term

Using Loop

```c
int triangle(int n)  
{  
it int total = 0;  
while (n > 0) // until n is 1  
{  
    total = total + n; // add n (column height) to total  
    --n; // decrement column height  
}  
return total;  
}
```
Finding $n^{th}$ Term

Using Loop

• The method cycles around the loop $n$ times, adding $n$ to total the first time, $n-1$ the second time, and so on down to 1, quitting the loop when $n$ becomes 0.

Finding the $n^{th}$ Term

• There’s another way to look at this problem.
• The value of the $n$th term can be thought of as the sum of only two things, instead of a whole series. These are
  1. The first (tallest) column, which has the value $n$.
  2. The sum of all the remaining columns.

Figure 6.3: Triangular number as column plus triangle
Finding $n^{\text{th}}$ Term

Using Loop

```c
int triangle(int n)
{
    int total = 0;
    while (n > 0)
    {
        total = total + n;
        --n;
    }
    return total;
}
```

Finding the Remaining Columns

If we knew about a method that found the sum of all the remaining columns, then we could write our `triangle()` method, which returns the value of the $n^{\text{th}}$ triangular number, like this:

```c
int triangle(int n)
{
    return( n + sumRemainingColumns(n) ); // (incomplete version)
}
```

But what have we gained here? It looks like it's just as hard to write the `sumRemainingColumns()` method as to write the `triangle()` method in the first place.
• Notice in Figure 6.3, however, that the sum of all the remaining columns for term $n$ is the same as the sum of all the columns for term $n-1$.

• Thus, if we knew about a method that summed all the columns for term $n$, we could call it with an argument of $n-1$ to find the sum of all the remaining columns for term $n$:

```c
int triangle(int n)
{
    return( n + sumAllColumns(n-1) ); // (incomplete version)
}
```

• But when you think about it, the `sumAllColumns()` method is doing exactly the same thing the `triangle()` method is doing:
  – summing all the columns for some number $n$ passed as an argument.
• So why not use the `triangle()` method itself, instead of some other method?
• That would look like this:

```c
int triangle(int n)
{
    return( n + triangle(n-1) ); // (incomplete version)
}
```

• It may seem amazing that a method can call itself, but why shouldn't it be able to?
• A method call is (among other things) a transfer of control to the start of the method.
• This transfer of control can take place from within the method as well as from outside.
Passing the Buck

• All this may seem like passing the buck.
  – Someone tells me to find the 9th triangular number.
  – I know this is 9 plus the 8th triangular number,
    • so I call Harry and ask him to find the 8th triangular number.
  – When I hear back from him, I'll add 9 to whatever he tells me, and that will be the answer.
  – Harry knows the 8th triangular number is 8 plus the 7th triangular number,
    • so he calls Sally and asks her to find the 7th triangular number.
  – This process continues with each person passing the buck to another one.

(Cont’d in the next slide)

Passing the Buck (Cont’d)

• Where does this buck-passing end?
• Someone at some point must be able to figure out an answer that doesn't involve asking another person to help them.
• If this didn't happen, there would be an infinite chain of people asking other people questions;
  – a sort of arithmetic Ponzi scheme that would never end.
  – In the case of triangle(), this would mean the method calling itself over and over in an infinite series that would paralyze the program.
The Buck Stops Here

- To prevent an infinite regress, the person who is asked to find the first triangular number of the series, when $n$ is 1, must know, without asking anyone else, that the answer is 1.
- There are no smaller numbers to ask anyone about, there's nothing left to add to anything else, so the buck stops there.
- We can express this by adding a condition to the `triangle()` method:

```java
int triangle(int n)
{
    if(n==1)
        return 1;
    else
        return( n + triangle(n-1) );
}
```

- The condition that leads to a recursive method returning without making another recursive call is referred to as the base case.
- It's critical that every recursive method have a base case to prevent infinite recursion and the consequent demise of the program.

The `triangle.java` Program

- Does recursion actually work?
- If you run the `triangle.java` program, you'll see that it does.
- Enter a value for the term number, $n$, and the program will display the value of the corresponding triangular number.
- Listing 6.1 (next slide) shows the `triangle.java` program.
The `main()` routine prompts the user for a value for `n`, calls `triangle()`, and displays the return value.

The `triangle()` method calls itself repeatedly to do all the work.

Here's some sample output:

```
Enter a number: 1000
Triangle = 500500
```

Incidentally, if you're skeptical of the results returned from `triangle()`, you can check them by using the following formula:

\[ \text{nth triangular number} = \frac{n^2+n}{2} \]
What's Really Happening?

Our observation
(between loop vs. recursion)

• If a loop is used, the method cycles around the loop \( n \) times, adding \( n \) to the total the first time, \( n-1 \) the second time and so on, down to 1, quitting the loop when \( n \) becomes 0.

• If recursion is used, then a base case is used that determines when the recursion ends.
Characteristics of Recursive Methods

- The recursive method calls itself to solve a smaller problem.
- The base case is the smallest problem that the routine solves and the value is returned to the calling method. (Terminal condition)

Is Recursion Efficient?

- Calling a method involves certain overhead in transferring the control to the beginning of the method and in storing the information of the return point.
- Memory is used to store all the intermediate arguments and return values on the internal stack.
- The most important advantage is that it simplifies the problem conceptually.

Mathematical Induction

- Recursion is programming equivalent of mathematical induction, which is a way of defining something in terms of itself.
- Using induction, we could define the triangular numbers mathematically by saying

  \[
  \text{if } n = 1 \\
  \text{tri}(n) = n + \text{tri}(n-1) \text{ if } n > 1
  \]

- Defining something in terms of itself may seem circular, but in fact it is perfectly valid (provided there is a base case).
Example (math induction)

• CPU burst prediction in scheduling

1. \( t_n \) = actual length of \( n^{th} \) CPU burst
2. \( \tau_{n+1} \) = predicted value for the next CPU burst
3. \( \alpha \), \( 0 \leq \alpha \leq 1 \)
4. Define:

\[
\tau_{n+1} = \alpha \ t_n + (1 - \alpha) \tau_n
\]

Prediction of the Length of the
Next CPU Burst

<table>
<thead>
<tr>
<th>CPU burst (( t_i ))</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; (( \tau_i ))</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Factorials

- Factorials are similar to triangle, addition being replaced by multiplication and the base case is when input is 0.
- The factorial of n is found by multiplying n with the factorial of n-1.
- E.g. factorial of $3 = 3 \times 2 \times 1$.
- Factorial of 0 is defined as 1.

Table 6.1: Factorials

<table>
<thead>
<tr>
<th>Number</th>
<th>Calculation</th>
<th>Factorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>by definition</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1 * 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2 * 1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3 * 2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4 * 6</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>5 * 24</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>6 * 120</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>7 * 720</td>
<td>5,040</td>
</tr>
<tr>
<td>8</td>
<td>8 * 5,040</td>
<td>40,320</td>
</tr>
<tr>
<td>9</td>
<td>9 * 40,320</td>
<td>362,880</td>
</tr>
</tbody>
</table>
• The factorial of 0 is defined to be 1. Factorial numbers grow large very rapidly, as you can see.
• A recursive method similar to `triangle()` can be used to calculate factorials. It looks like this:

```java
int factorial(int n)
{
    if(n==0)
        return 1;
    else
        return (n * factorial(n-1) );
}
```
• There are only two differences between `factorial()` and `triangle()`.
  – First, `factorial()` uses an `*` instead of a `+` in the expression `n * factorial(n-1)`.
  – Second, the base condition occurs when `n` is 0, not 1.

**Figure 6.5:** The recursive `factorial()` method
Anagrams

- Anagramming a word is arranging letters of the word in different orders.
- E.g: Anagramming cat gives: cat, cta, atc, act, tca, tac.
  - See Table 6.2
- The strategy to achieve this is as follows:
  -- The number of possibilities is the factorial of the number of letters.
  -- The rightmost n-1 letters are anagrammed.
  -- Rotate all n letters, shifting all the letters one position left except for the leftmost letter which rotates back to the right.
  -- Repeat these steps n times.
- Rotating the word gives each letter the chance to begin the word.
- While the selected letter occupies the first position, all the other letters are then anagrammed.

Anagrams (Cont’d.)

- We must rotate back to the starting point with two letters before performing a three-letter rotation.
- The rightmost n-1 letters are anagrammed by recursion.
- The base case occurs when the size of the word to be anagrammed is only one letter.

- ReaderFiles\Chap06\anagram\anagram.java
Recursive Binary Search

- Binary search can also be implemented using recursion.
- The method can call itself with new starting and ending values.
- The base case is when the starting value is greater than the end value.
Divide-and-Conquer

- Recursive binary search is an example of divide-and-conquer.
- The idea is to divide the bigger problem into smaller problems and solve each one separately.
- The solution to each smaller problem is to divide it into even smaller problems and solve them.
- The process continues till the base case is reached.
- It can be used with recursion as well as non-recursion.

ReaderFiles\Chap06\binarySearch\binarySearch.java (next slide)
private int recFind(long searchKey, int lowerBound, 
int upperBound) 
{
int curIn;

curIn = (lowerBound + upperBound) / 2;
if(a[curIn]==searchKey) 
    return curIn;              // found it
else if(lowerBound > upperBound) 
    return nElems;             // can't find it
else                          // divide range
{
    if(a[curIn] < searchKey)   // it's in upper half 
        return recFind(searchKey, curIn+1, upperBound);
    else                       // it's in lower half 
        return recFind(searchKey, lowerBound, curIn-1);
}  // end else divide range
}  // end recFind()

//-----------------------------------------------------------
public void insert(long value)    // put element into array 
{
    int j;
    for(j=0; j<nElems; j++)        // find where it goes
        if(a[j] > value)            // (linear search)
            break;
    for(int k=nElems; k>j; k--)    // move bigger ones up
        a[k] = a[k-1];
    a[j] = value;                  // insert it
    nElems++;                      // increment size
}  // end insert()

//-----------------------------------------------------------
public void display()             // displays array contents
{
    for(int j=0; j<nElems; j++)    // for each element,
    System.out.print(a[j] + " "); // display it
    System.out.println("");
}  // end class ordArray
class BinarySearchApp
{
    public static void main(String[] args)
    {
        int maxSize = 100;            // array size
        ordArray arr;                 // reference to array
        arr = new ordArray(maxSize);  // create the array
        arr.insert(72);               // insert items
        arr.insert(90);
        arr.insert(45);
        arr.insert(126);
        arr.insert(54);
        arr.insert(99);
        arr.insert(144);
        arr.insert(27);
        arr.insert(135);
        arr.insert(81);
        arr.insert(18);
        arr.insert(108);
        arr.insert(9);
        arr.insert(117);
        arr.insert(63);
        arr.insert(36);
        arr.display();                // display array
        int searchKey = 27;            // search for item
        if( arr.find(searchKey) != arr.size() )
            System.out.println("Found " + searchKey);
        else
            System.out.println("Can't find " + searchKey);
    }  // end main()
}  // end class BinarySearchApp

---

Towers of Hanoi

- A problem that consists of a number of disks placed on three columns.
- The disks have different diameters and holes in the middle so they will fit over the columns.
- All the disks start out on the first column.
- The object of the problem is to transfer all the disks from first column to last.
- Only one disk can be moved at a time and no disk can be placed on a disk that’s smaller than itself.
Workshop Applet: The Towers

There are three ways to use the workshop applet.

• You can attempt to solve the puzzle manually, by dragging the disks from tower to tower.

• You can repeatedly press the Step button to watch the algorithm solve the puzzle. At each step in the solution, a message is displayed, telling you what the algorithm is doing.

• You can press the Run button and watch the algorithm solve the puzzle with no intervention on your part; the disks zip back and forth between the posts.

Moving Subtrees

• Let's call the initial tree-shaped (or pyramid-shaped) arrangement of disks on tower A a tree.

• As you experiment with the applet, you'll begin to notice that smaller tree-shaped stacks of disks are generated as part of the solution process. Let's call these smaller trees, containing fewer than the total number of disks, subtrees.
  – For example, if you're trying to transfer 4 disks, you'll find that one of the intermediate steps involves a subtree of 3 disks on tower B, as shown below:

• These subtrees form many times in the solution of the puzzle. This is because the creation of a subtree is the only way to transfer a larger disk from one tower to another:
  – all the smaller disks must be placed on an intermediate tower, where they naturally form a subtree.
A Rule Of Thumb

- If the subtree you're trying to move has an odd number of disks, start by moving the topmost disk directly to the tower where you want the subtree to go.
- If you're trying to move a subtree with an even number of disks, start by moving the topmost disk to the intermediate tower.

Towers of Hanoi: Solution

- Assume that you want to move all the disks from a source tower, S to a destination tower, D.
- Assume an intermediate tower, I.
- Assume n disks on S.
  1. Move the subtree consisting of the top n-1 disks from S to I.
  2. Move the remaining (largest) disk from S to D.
  3. Move the subtree from I to D.
Figure 6.13: Recursive solution to Towers puzzle
Mergesort

- The heart of mergesort algorithm is the merging of two already sorted arrays.
- Merging two sorted arrays A and B creates a third array C that contains all the elements of A and B arranged in sorted order.
- The idea is to divide an array in half, sort each half, and then use the merge method to merge the two halves into a single sorted array.
- To sort the half arrays, call the sorting method recursively, dividing the arrays into quarters, so on and so forth.

```java
// merge.java
// demonstrates merging two arrays into a third
// to run this program: C>java MergeApp

class MergeApp {
    public static void main(String[] args) {
        int[] arrayA = {23, 47, 81, 95};
        int[] arrayB = {7, 14, 39, 55, 62, 74};
        int[] arrayC = new int[10];

        merge(arrayA, 4, arrayB, 6, arrayC);
        display(arrayC, 10);
    }
}
```
// merge A and B into C
public static void merge( int[] arrayA, int sizeA,
        int[] arrayB, int sizeB,
        int[] arrayC )
{
    int aDex=0, bDex=0, cDex=0;

    while(aDex < sizeA && bDex < sizeB)  // neither array empty
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];

    while(aDex < sizeA)                  // arrayB is empty,
        arrayC[cDex++] = arrayA[aDex++];  // but arrayA isn't

    while(bDex < sizeB)                  // arrayA is empty,
        arrayC[cDex++] = arrayB[bDex++];  // but arrayB isn't
}  // end merge()

// display array
public static void display(int[] theArray, int size)
{
    for(int j=0; j<size; j++)
        System.out.print(theArray[j] + " ");
    System.out.println(" ");
}  // end display()

// end class MergeApp

The MERGESORT Workshop Applet
private void recMergeSort(long[] workSpace, int lowerBound,
                      int upperBound)
{
    if(lowerBound == upperBound)            // if range is 1,       // no use sorting
        return;
    else                                    // find midpoint
        {                                     // sort low half
            int mid = (lowerBound+upperBound) / 2;
            recMergeSort(workSpace, lowerBound, mid);  // sort high half
            recMergeSort(workSpace, mid+1, upperBound);
            merge(workSpace, lowerBound, mid+1, upperBound);       // merge them
        }   // end else
}   // end recMergeSort()

// Where is the complete mergesort listing?

Mergesort: Efficiency

- Mergesort runs in $O(N \times \log N)$ time.
  - Number of copies
  - Number of comparisons
- Assuming that the number of items is a power of 2
  - the number of copies is proportional to $N \times \log_2 N$
  - for each individual merging operation
    - the maximum number of comparisons is one less than the number of items being merged.
    - the minimum number of comparisons is always half the number of items being merged.
Eliminating Recursion

- Some algorithms use recursive methods, some don’t.
  - The recursive `triangle()` and `factorial()` methods can be implemented more efficiently using a simple loop.
  - However, various divide-and-conquer algorithms, such as mergesort, work very well as a recursive routine.
- Often an algorithm is easy to conceptualize as a recursive method, but in practice the recursive method might prove to be inefficient.
  - In such cases, it is useful to transform the recursive approach into a non-recursive approach.
  - Such transformation makes use of stack.

Recursion and Stacks

- Most compilers implement recursion using stacks.
- When a method is called, the compiler pushes the arguments to the method and the return address on the stack and then transfers the control to the method.
- When the method returns, it pops these values off the stack.
- The arguments disappear and the control returns to the return address.
Simulating a Recursive Method

• How any recursive solution can be transformed into a stack-based solution?
• Remember the recursive `triangle()` method:
  ```java
  int triangle(int n)
  {
    if(n==1)
      return 1;
    else
      return( n + triangle(n-1) );
  }
  ```

Simulating a Recursive Method

• We’re going to break `triangle()` algorithm down into its individual operations, making each operation one case in a switch statement. (Java doesn’t support `goto`.)
• The switch statement is enclosed in a method called `step()`. Each call to `step()` causes one case section within the switch to be executed.
• Calling `step()` repeatedly will eventually execute all the code in the algorithm.
Simulating a Recursive Method

- Listing 6.7 The stackTriangle.java Program
  - In stackTriangle.java we have a program that more or less systematically transforms a program that uses recursion into a program that uses a stack.
  - This suggests that such a transformation is possible for any program that uses recursion, and in fact this is the case.

Simulating a Recursive Method

- With some additional work, you can systematically refine the code in stackTriangle.java, simplifying it and even eliminating the switch statement entirely to make the code more efficient.
- In practice, however, it’s usually more practical to rethink the algorithm from the beginning, using a stack-based approach instead of a recursive approach.
  - Listing 6.8 stackTriangle2.java shows what happens when we do that with the triangle() method.
Precision & Science First, Art Next

• Two short while loops in the stackTriangle() method substitute, in stackTriangle2.java, for the entire step() method of the stackTriangle.java program.
  – Of course, in this program you can see by inspection that you can eliminate the stack entirely and use a simple loop.
  – However, in more complicated algorithms the stack must remain.

• Often you’ll need to experiment to see whether
  – a recursive method,
  – a stack-based approach, or
  – a simple loop
  is the most efficient (or practical) way to handle a particular situation.

Recursive Applications

• Raising a Number to a Power
• The Knapsack problem: (Next slide)
  – This involves trying to fit items of different weights into a knapsack so that the knapsack ends up with a specified total weight.

• Combinations: Picking a Team.
  – A combination is a selection of things in which their order doesn’t matter.
Knapsack Problem

Algorithm:
- Start by selecting the first item. The remaining items must add up to the knapsack’s target weight minus the first item; this is a new target weight.
- Try one by one, each of the possible combinations of the remaining items.
- If none of the combinations work, discard the first item, and start the whole process again with the second item.
- Continue this with the third item and so on until you’ve tried all the combinations at which point you know there is no solution.
- If at any point in this process the sum of the items you selected adds up to the target, you are done.