Chapter 7

Recursion
Recursive evaluation of the sum of the first $N$ integers

```java
// Evaluate the sum of the first n integers
public static long s(int n)
{
    if (n == 1)
        return 1;
    else
        return s(n - 1) + n;
}
```
A recursive routine for printing $N$ in decimal form

```java
// Print n in base 10, recursively.
// Precondition: n >= 0.
public static void printDecimal( long n )
{
    if( n >= 10 )
        printDecimal( n / 10 );
    System.out.print( (char) ('0' + ( n % 10 ) ) );
}
```
private static final String DIGIT_TABLE = "0123456789abcdef";

// Print n in any base, recursively.
// Precondition: n >= 0, base is valid.
public static void printInt( long n, int base )
{
    if( n >= base )
        printInt( n / base, base );
    System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
}
public final class PrintInt
{
    private static final String DIGIT_TABLE = "0123456789abcdef";
    private static final int MAX_BASE = DIGIT_TABLE.length();

    // Print n in any base, recursively
    // Precondition: n >= 0, 2 <= base <= MAX_BASE
    private static void printIntRec( long n, int base )
    {
        if( n >= base )
            printIntRec( n / base, base );
        System.out.print( DIGIT_TABLE.charAt( (int)( n % base ) ) );
    }

    // Driver routine
    public static void printInt( long n, int base )
    {
        if( base <= 1 || base > MAX_BASE )
            System.err.println( "Cannot print in base " + base );
        else
        {
            if( n < 0 )
            {
                n = -n;
                System.out.print( "-" );
            }
            printIntRec( n, base );
        }
    }
}
figure 7.5
A stack of activation records
A recursive routine for Fibonacci numbers: A bad idea

```java
// Compute the Nth Fibonacci number.
// Bad algorithm.
public static long fib( int n )
{
    if( n <= 1 )
        return n;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```
A trace of the recursive calculation of the Fibonacci numbers.
**Figure 7.8**
A tree viewed recursively.
A tree
// Evaluate n!
public static long factorial(int n)
{
    if (n <= 1)  // base case
        return 1;
    else
        return n * factorial(n - 1);
}
```java
/**
 * Performs the standard binary search using two comparisons
 * per level. This is a driver that calls the recursive method.
 * @return index where item is found or NOT_FOUND if not found.
 */
public static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType[] a, AnyType x )
{
    return binarySearch( a, x, 0, a.length - 1 );
}

/**
 * Hidden recursive routine.
 */
private static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType[] a, AnyType x, int low, int high )
{
    if( low > high )
        return NOT_FOUND;

    int mid = ( low + high ) / 2;

    if( a[ mid ].compareTo( x ) < 0 )
        return binarySearch( a, x, mid + 1, high );
    else if( a[ mid ].compareTo( x ) > 0 )
        return binarySearch( a, x, low, mid - 1 );
    else
        return mid;
}
```
Figure 7.12
A recursively drawn ruler
// Java code to draw Figure 7.12.
void drawRuler( Graphics g, int left, int right, int level )
{
    if( level < 1 )
        return;

    int mid = ( left + right ) / 2;

    g.drawLine( mid, 80, mid, 80 - level * 5 );

drawRuler( g, left, mid - 1, level - 1 );
drawRuler( g, mid + 1, right, level - 1 );
}
**figure 7.14**

(a) A fractal star outline drawn by the code shown in Figure 7.15; (b) The same star immediately before the last square is added.
figure 7.15

Code for drawing the fractal star outline shown in Figure 7.14

```java
// Draw picture in Figure 7.14.
void drawFractal( Graphics g, int xCenter,
                 int yCenter, int boundingDim )
{
    int side = boundingDim / 2;

    if( side < 1 )
        return;

    // Compute corners.
    int left =  xCenter - side / 2;
    int top  =  yCenter - side / 2;
    int right = xCenter + side / 2;
    int bottom = yCenter + side / 2;

    // Recursively draw four quadrants.
    drawFractal( g, left, top, boundingDim / 2 );
    drawFractal( g, left, bottom, boundingDim / 2 );
    drawFractal( g, right, top, boundingDim / 2 );
    drawFractal( g, right, bottom, boundingDim / 2 );

    // Draw central square, overlapping quadrants.
    g.fillRect( left, top, right - left, bottom - top );
}
```
/**
 * Return x^n (mod p)
 * Assumes x, n >= 0, p > 0, x < p, 0^0 = 1
 * Overflow may occur if p > 31 bits.
 */

public static long power( long x, long n, long p )
{
    if( n == 0 )
        return 1;

    long tmp = power( ( x * x ) % p, n / 2, p );

    if( n % 2 != 0 )
        tmp = ( tmp * x ) % p;

    return tmp;
}
figure 7.17

Computation of greatest common divisor

```java
/**
 * Return the greatest common divisor.
 */

public static long gcd(long a, long b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```
// Internal variables for fullGcd
private static long x;
private static long y;

/**
 * Works back through Euclid's algorithm to find
 * x and y such that if gcd(a,b) = 1,
 * ax + by = 1.
 */
private static void fullGcd( long a, long b )
{
    long x1, y1;
    if( b == 0 )
    {
        x = 1;
        y = 0;
    }
    else
    {
        fullGcd( b, a % b );
        x1 = x; y1 = y;
        x = y1;
        y = x1 - ( a / b ) * y1;
    }
}

/**
 * Solve ax == 1 (mod n), assuming gcd( a, n ) = 1.
 * @return x.
 */
public static long inverse( long a, long n )
{
    fullGcd( a, n );
    return x > 0 ? x : x + n;
### Running Sum from the Center

<table>
<thead>
<tr>
<th>First Half</th>
<th>Second Half</th>
<th>Values</th>
<th>Running sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7*</td>
<td>7*</td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

*denotes maximum for each half.*
figure 7.20
A divide-and-conquer algorithm for the maximum contiguous subsequence sum problem

/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Driver for divide-and-conquer maximum contiguous subsequence sum algorithm.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
}
Figure 7.21
Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm for \(N = 8\) elements
Figure 7.22
Some of the subproblems solved recursively in Figure 7.23
// Return minimum number of coins to make change.
// Simple recursive algorithm that is very inefficient.
public static int makeChange( int [] coins, int change )
{
    int minCoins = change;

    for( int i = 0; i < coins.length; i++ )
        if( coins[ i ] == change )
            return 1;

    // No match; solve recursively.
    for( int j = 1; j <= change / 2; j++ )
    {
        int thisCoins = makeChange( coins, j )
                        + makeChange( coins, change - j );
        if( thisCoins < minCoins )
            minCoins = thisCoins;
    }

    return minCoins;
}
**Figure 7.24**
An alternative recursive algorithm for the coin-changing problem
// Dynamic programming algorithm to solve change-making problem.
// As a result, the coinsUsed array is filled with the
// minimum number of coins needed for change from 0 -> maxChange
// and lastCoin contains one of the coins needed to make the change.
public static void makeChange( int[] coins, int differentCoins,
    int maxChange, int[] coinsUsed, int[] lastCoin )
{
    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;

    for( int cents = 1; cents <= maxChange; cents++ )
    {
        int minCoins = cents;
        int newCoin = 1;

        for( int j = 0; j < differentCoins; j++ )
        {
            if( coins[ j ] > cents )  // Cannot use coin j
                continue;
            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins )
            {
                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin = coins[ j ];
            }
        }
        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ] = newCoin;
    }
}

**figure 7.25**
A dynamic programming algorithm for solving the change-making problem by computing
optimal change for all amounts from 0 to maxChange and maintaining information to construct
the actual coin sequence.
final class Best
{
  int row;
  int column;
  int val;

  public Best( int v )
  { this( v, 0, 0 ); }

  public Best( int v, int r, int c )
  { val = v; row = r; column = c; }
}
class TicTacToe
{
    public static final int HUMAN = 0;
    public static final int COMPUTER = 1;
    public static final int EMPTY = 2;
    public static final int HUMAN_WIN = 0;
    public static final int DRAW = 1;
    public static final int UNCLEAR = 2;
    public static final int COMPUTER_WIN = 3;

    // Constructor
    public TicTacToe()
    { clearBoard(); }

    // Find optimal move
    public Best chooseMove( int side )
    { /* Implementation in Figure 7.29 */ }

    // Compute static value of current position (win, draw, etc.)
    private int positionValue()
    { /* Implementation in Figure 7.28 */ }

    // Play move, including checking legality
    public boolean playMove( int side, int row, int column )
    { /* Implementation in online code */ }

    // Make board empty
    public void clearBoard()
    { /* Implementation in online code */ }

    // Return true if board is full
    public boolean boardIsFull()
    { /* Implementation in online code */ }

    // Return true if board shows a win
    public boolean isAWin( int side )
    { /* Implementation in online code */ }

    // Play a move, possibly clearing a square
    private void place( int row, int column, int piece )
    { board[ row ][ column ] = piece; }

    // Test if a square is empty
    private boolean squareIsEmpty( int row, int column )
    { return board[ row ][ column ] == EMPTY; }

    private int [ ] [ ] board = new int[ 3 ][ 3 ];
}
figure 7.28
Supporting routine for evaluating positions

```java
1 // Compute static value of current position (win, draw, etc.)
2 private int positionValue( )
3 {
4     return isAWin( COMPUTER ) ? COMPUTER_WIN :
5         isAWin( HUMAN ) ? HUMAN_WIN :
6         boardIsFull( ) ? DRAW : UNCLEAR;
7 }
```
// Find optimal move
public Best chooseMove( int side )
{
    int opp;       // The other side
    Best reply;    // Opponent's best reply
    int dc;        // Placeholder
    int simpleEval; // Result of an immediate evaluation
    int bestRow = 0;
    int bestColumn = 0;
    int value;

    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
        return new Best( simpleEval );

    if( side == COMPUTER )
    {
        opp = HUMAN; value = HUMAN_WIN;
    }
    else
    {
        opp = COMPUTER; value = COMPUTER_WIN;
    }

    for( int row = 0; row < 3; row++ )
        for( int column = 0; column < 3; column++ )
            if( squareIsEmpty( row, column ) )
            {
                place( row, column, side );
                reply = chooseMove( opp );
                place( row, column, EMPTY );

                // Update if side gets better position
                if( side == COMPUTER && reply.val > value
                    || side == HUMAN && reply.val < value )
                {
                    value = reply.val;
                    bestRow = row; bestColumn = column;
                }
            }

    return new Best( value, bestRow, bestColumn );
}
figure 7.30
Grid for Exercise 7.31