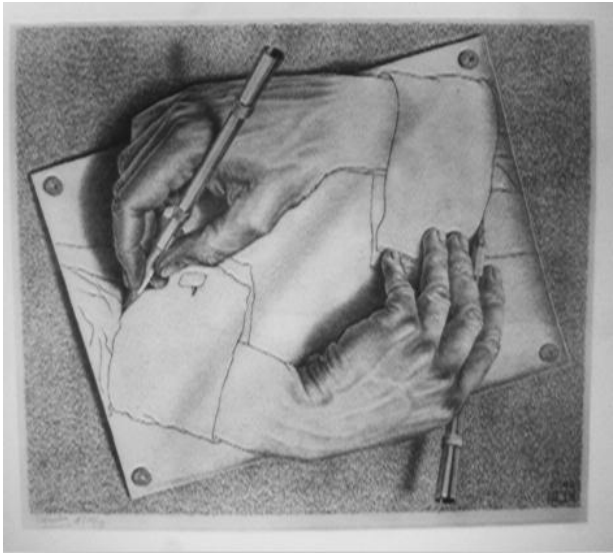
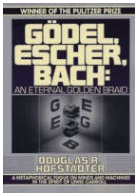
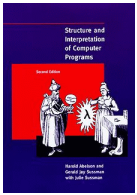


Drawing Hands is a lithograph by the Dutch artist M. C. Escher first printed in January 1948.

It is referenced in the Pulitzer Prize winning book *Gödel, Escher, Bach*, by Douglas Hofstadter, who calls it an example of a strange loop.

It is also used in *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman as an allegory for the eval and apply functions of programming language interpreters.

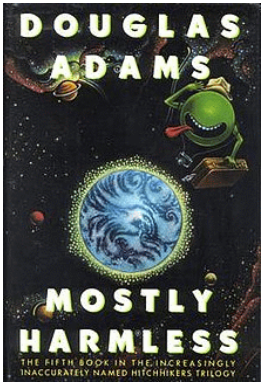




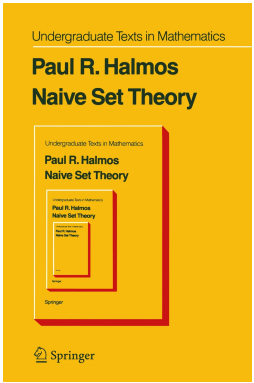
Recursion

Anything that, in happening, causes itself to happen again, happens again.

Douglas Adams, *Mostly Harmless*



Recursion



"Droste effect" in art

Reading

Sedgwick & Wayne, *Introduction to Programming in java*, Section 2.4.

Horstman, *Java Essentials*, Chapter 17, page 667.

Horstman, *Big Java*, Chapter 18, page 653.

Adams, Nyhoff, Nyhoff, *Java*, Section 8.6, page 457.

Skansholm, *Java From the Beginning*, Section 15.4, page 488.

Main, *Data Structures Using Java*, Chapter 8, page 371.

Iteration

The typical computer has an efficient "goto" or jump instruction which is used to implement iteration (the for, for-each, while, and do-while statements).

Using iterations appears to be natural to most programmers. Programmers find it easier to trace the actions of a simple machine than express the complete meaning of a loop.

Reading, writing, and reasoning with iteration is difficult because it envolves time. (See loop invariants.)

Recursion

Self-reference appears often in data structures and in computation.

- Lists data structures
- Mergesort, quicksort, greatest common divisor, fast Fourier transform
- The file system where a folder contains files and other folders

Recursion

To understand and create solutions to complex problems it is necessary to decomposed them into smaller problems. When the smaller problems has the same structure as the bigger problem, recursion is a big win.

In these cases, the smaller problem is fundamentally the same as the original problem, when the right abstraction or parameterization has been identified. A recursive solution has been found.

Recursion

[Realization: until one encounters trees structures, the most important recursive structures are *linear*. Linear strutures are easy to address with iteration, hence recursion seems unfamiliar and unnecessary.]

Numbers	$n, n-1, \quad \ldots, \quad 2, \quad 1, \quad 0$	$n \leftarrow n-1$
Strings	$\ldots$, `"aa"`, `"a"`, `""`	`s` $\leftarrow$ `s.substring(1)`
Lists	$\ldots$, `of(a,a)`, `of(a)`, `of()`	`l` $\leftarrow$ `l.subList(1,l.size())`

[Sadly, the linear recursive stucture that numbers, strings, and lists have in common is not clearly manifested in Java. See Haskell.]

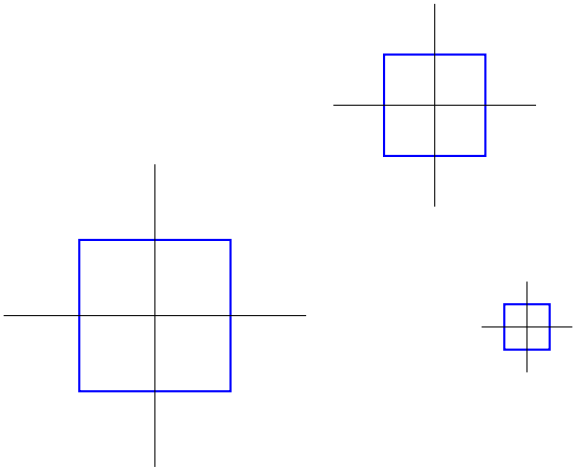
Recursion

A recursive solution has the advantage of reusing the same technique (with different size inputs) and so fewer subprocedures need be written.

Recursion is a powerful and elegant way of solving problems.

Sometimes a recursive solution is easier to create than an iterative one even if one does not see it a first.

Parameterization



Drawing a square parameterized by the orgin of the coordinate system and by the length of the side.

Iteration and Recursion

Anything one can do with iteration, one can do with recursion.

Anything one can do with recursion, one can do with iteration.

Recursion

A **recursive call** is a place in the program in which a subprogram may call itself.

(When tracing the dynamic execution of the program we may also speak of a recursive call.)

Anatomy of recursion:

- Base case. A simple case solved immediately without requiring recursion.
- General case. A case for many inputs solved with recursion.
- Smaller-caller issue. Each recursive call must involve a "smaller" problem or one "closer" to the base case.

There are many kinds of recursive programs based on the difference kinds of structures we compute with. This is closely related to proof by induction.

Know When to Stop

A: Guten Appetit!

B: Danke, gleichfalls

A: Danke, dass du dich bedankt hast

B: Keine Ursache, bitte und Danke für das Danke

A: Ist doch selbstverständlich, Danke

B: Na so selbstverständlich auch nicht, deshalb : Vielen Dank

A: Danke, wie nett

B: Kein Problem, also Danke und guten Appetit

A: Das wird nichts. Essen ist kalt und Pause rum, aber vielen Dank für die Unterhalten

B: Gern geschehen, Danke! *MIST! Nächtesmal sag' ich nichts!!*

Know When to Stop

Recursive Definitions of Functions in Mathematics

Lots of recursive definitions are found in mathematics. That is because the natural numbers are a recursive data structures.

Given a recursive definition is often easy to translate into code.

(But caution: many definitions were designed without computational efficiency in mind.)

Writing computationally efficient solutions requires skill, just as writing computationally efficient iterative solutions.

Do not blame recursion (or iteration) for poor design.

Recursive Definitions of Functions

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ x \cdot x^{(n-1)} & \text{otherwise} \end{cases}$$

$$fib(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Recursive Definitions of Functions

Greatest common denominator: ror $x \geq y$,

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0, \\ gcd(y, x \bmod y) & \text{otherwise} \end{cases}$$

Binomial coefficient: for $0 \leq r \leq n$,

$$C(n, r) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = r, \\ C(n-1, r-1) + C(n-1, r) & \text{otherwise} \end{cases}$$

Recursive Definitions of Functions

Sterling numbers of the first kind (permutations with k disjoint cycles): for $0 \leq k \leq n$,

$$s(n,k) = \begin{cases} 1 & \text{if } n = k, \\ 0 & \text{if } k = 0, \\ s(n-1,k-1) + (n-1) \cdot s(n-1,k) & \text{otherwise} \end{cases}$$

Sterling numbers of the second kind (partions with k distinct subsets): for $0 \leq k \leq n$,

$$S(n,k) = \begin{cases} 1 & \text{if } n = k, \\ 0 & \text{if } k = 0, \\ S(n-1,k-1) + k \cdot S(n-1,k) & \text{otherwise} \end{cases}$$

McCarthy's 91 function

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100, \\ M(M(n + 11)) & \text{otherwise} \end{cases}$$

$$\begin{array}{rcll} M(87) & = & M(M(98)) & 87 \leq 100 \\ & & M(M(M(109))) & 98 \leq 100 \\ & & M(M(99)) & 109 > 100 \\ & & M(M(M(110))) & 99 \leq 100 \\ & & M(M(100)) & 110 > 100 \\ & & 91 & 101 > 100 \end{array}$$

We need more practice tracing.

Recursive Definitions of Functions

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ Ack(m-1, Ack(m,n-1)) & \text{otherwise} \end{cases}$$

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

```
fact(0) = 1
fact(n) = n*fact(n-1)
```

```
public class Main
   public static int fact (final int n) {
     if (n==0) return 1;
     else return n * fact(n-1);
   }
   public static void main (final String[] args) {
     System.out.println (fact(4));
     System.out.println (fact(7));
   }
}
```

Alternate Function Definitions

```
int fact (final int n) {
   assert 0<=n;
   if (n==0) return 1;
   else return n * fact(n-1);
}

int fact (final int n) {
   assert 0<=n;
   return (n==0) ? 1 : n*fact(n-1);
}

int fact (final int n) {
   assert 0<=n;
   return switch(n){case 0->1; default->n*fact(n-1);}
}
```

[Java has no data type for natural numbers (non-negative integers).]

Alternate Function Definitions

One might accept a version with an unnecessary name:

```
int fact (final int n) {
   final int ret;
   if (n==0) ret=1;
   else ret = n * fact(n-1);
   return ret;
}
```

```
int fact (final int n) {
   final int ret = n==0? 1 : n*fact(n-1);
   return ret;
}
```

But never the error prone:

```
int fact (final int n) {
   int ret=1;  // Can't be final!
   if (n>0) ret = n * fact(n-1);
   return ret;
}
```

Factorial

To correctly capture the idea of a recursive solution:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

we realize the following identities hold:

```
fact(0) = 1              -- n=0
fact(n) = n*fact(n-1)    -- n>0
```

which is clearly realized in the Java code:

```
int fact (final int n) {
   assert 0<=n;
   if (n==0) return 1;
   else return n * fact(n-1);
}
```

Caution: not efficient.

Exponential (Java)

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ x \cdot x^{(n-1)} & \text{otherwise} \end{cases}$$

```
pow(_,0) = 1
pow(x,n) = x*pow(x,n-1)
```

```
int pow (final int x, final int n) {
   assert 0<=n;
   if (n==0) return 1;
   else return x * pow (x, n-1);
}
```

Caution: not efficient.

Exponential (Java)

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ x^{n/2} \cdot x^{n/2} & \text{if } n \text{ divisible by 2}, \\ x \cdot x^{(n-1)} & \text{otherwise} \end{cases}$$

```
pow(x,n)
    | n=0         = 1
    | even n      = let y=pow(x,n `div` 2) in y*y
    | otherwise = x*pow(x,n-1)
```

```
int pow (final int x, final int n) {
   assert 0<=n;
   if (n==0) return 1;
   else if (n%2==0) {
     final int y = pow(x,n/2); return y*y;
   } else return x * pow (x, n-1);
}
```

Fibonacci (Java)

$$fib(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

```
fib n =
    | n<=2       = 1
    | otherwise = fib(n-1) + fib(n-2)
```

```
int fib (final int n) {
  if (n==1) return 1;
  else if (n==2) return 1;
  else return fib(n-1) + fib(n-2)
}
```

Caution: not efficient.

GCD (Java)

```
gcd (p,0) = p
gcd (p,q) = gcd (q, p `mod` q)
```

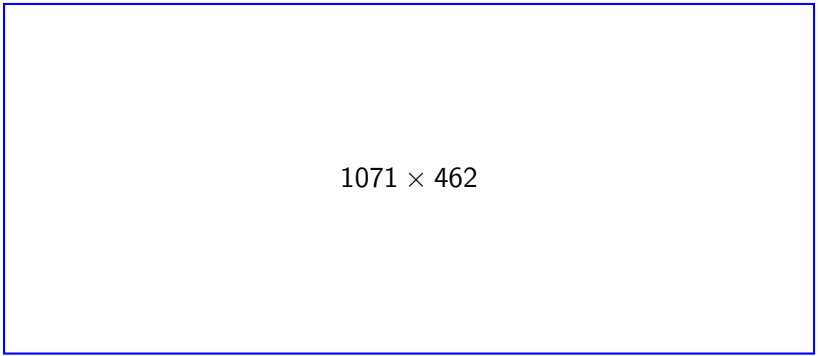
```
int gcd (final int p, final int q) {
  if (q==0) return p;
  else return gcd (q, p%q);
}
```

Tail recursive: very efficient.

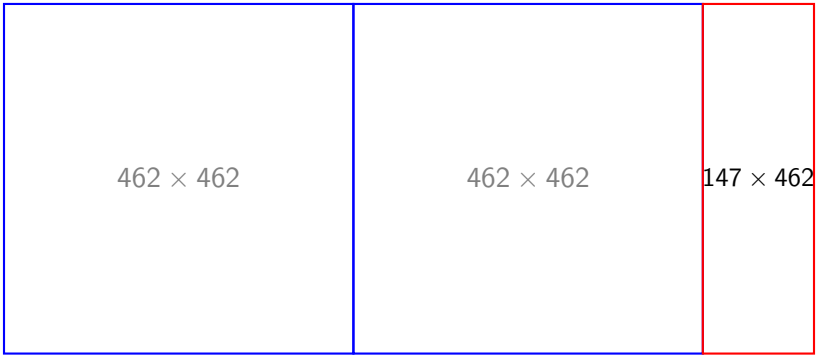
Discovering GCD

One way of visualizing the recursive step is by perfectly tiling a p by q rectangle with square tiles. The only square tiles which will work are those with side lengths which are a common divisor of both p and q.
The recursion emerges when we get the insight that the original problem of tiling a large rectangle can be reduced to the same problem of tiling a smaller rectangle.

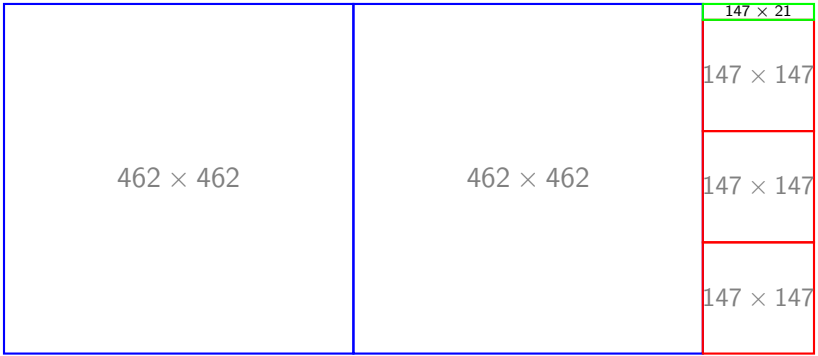
GCD of 1071 and 462 is 21



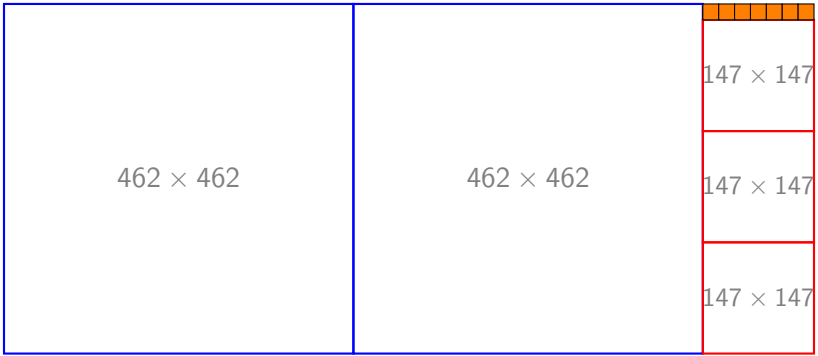
GCD of 1071 and 462 is 21



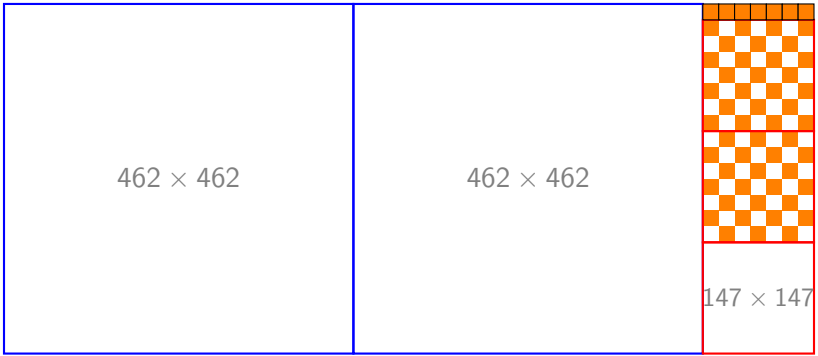
GCD of 1071 and 462 is 21



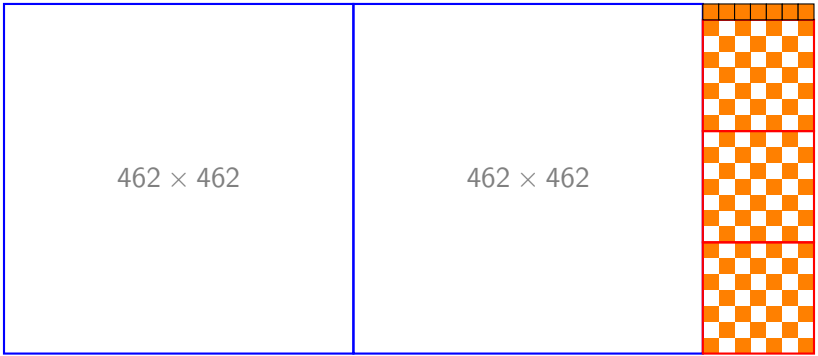
GCD of 1071 and 462 is 21



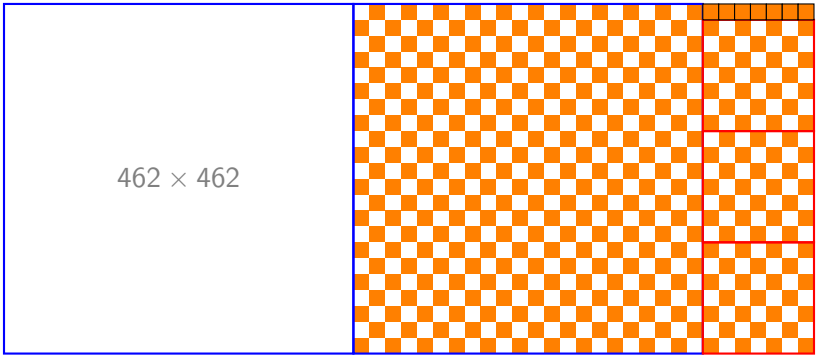
GCD of 1071 and 462 is 21



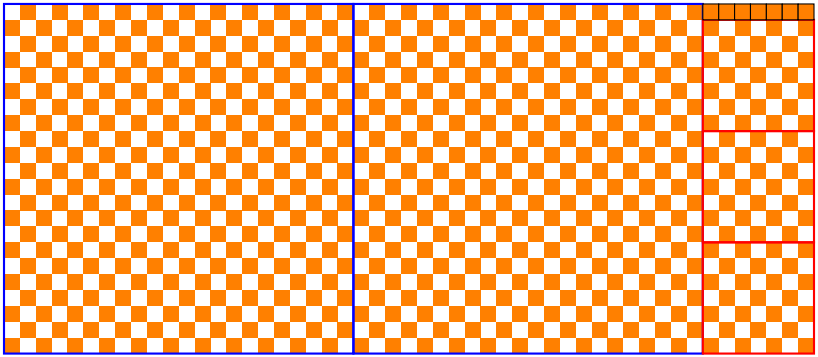
GCD of 1071 and 462 is 21



GCD of 1071 and 462 is 21



GCD of 1071 and 462 is 21



Binomial coefficient (Java)

For $0 \leq r \leq n$,

$$C(n,r) = \begin{cases} 1 & \text{if } r = 0 \text{ or } r = n, \\ C(n-1,r-1) + C(n-1,r) & \text{otherwise} \end{cases}$$

```
c(n,r)
  | r==0 || r==n = 1
  | otherwise    = c(n-1,r-1) + c(n-1,r)
```

```
int binomial (final int n, final int r) {
   if (r==0 || r==n) return 1;
   else return binomial (n-1,r-1) +
      binomial (n-1,r);
}
```

Stirling Numbers of the First Kind

For $0 \leq k \leq n$,

$$s(n,k) = \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } k = n, \\ s(n-1,k-1) + (n-1) \cdot s(n-1,k) & \text{otherwise} \end{cases}$$

```
s(n,k)
  | k==0    = 0
  | k==n    = 1
  | otherwise = s(n-1,k-1) + (n-1)*s(n-1,k)
```

```
int stirling1 (final int n, final int k) {
   if (k==0) return 0;
   else if (n==k) return 1;
   else return stirling1 (n-1,k-1) +
      (n-1)*stirling1(n-1,k);
}
```

Stirling Numbers of the Second Kind

For $0 \leq k \leq n$,

$$S(n,k) = \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } n = k, \\ S(n-1,k-1) + k \cdot S(n-1,k) & \text{otherwise} \end{cases}$$

```
s(n,k)
  | k==0    = 0
  | k==n    = 1
  | otherwise = S(n-1,k-1) + k*S(n-1,k)
```

```
int stirling2 (final int n, final int k) {
   if (n==k) return 1;
   else if (k==0) return 0;
   else return stirling2 (n-1,k-1) +
      k*stirling2(n-1,k);
}
```

Ackermann Function

A well-known, fast-growing function.

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ Ack(m-1,1) & \text{if } n = 0, \\ Ack(m-1, Ack(m,n-1)) & \text{otherwise} \end{cases}$$

```
ack(m,n)
   | m==0       = n+1
   | m==0       = ack(m-1,1)
   | otherwise = ack (m-1, ack(m,n-1))
```

```
int ackermann (final int m, final int n) {
   if (m==0) return n+1;
   else if (n==0) return ackermann (m-1,1);
   else return ackermann (m-1,ackermann (m,n-1));
}
```

Ackermann Function

	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	$2^{2^{2^{65536}}} - 3$
5	65533				

A small recursive program can do a lot of computation!

More Recursive Functions

Recursion is not just for natural numbers!

Strings, lists, and arrays by length; algebraic data structures by subterms.

More Recursive Functions

```
String deleteBlanks (final String x) {
  if (x.isEmpty()) return x;
  else if (x.charAt(0)==' ') return deleteBlanks (x.substring
  else return x.charAt(0)+deleteBlanks (x.substring(1));
}
```

This follows the elegant recursive nature of strings. Also substring is does not require copying because strings are immutable in Java. However the function is not tail-recursive. Not very suitable in Java.

```
x.replaceAll (" ", "");
x.replaceAll ("\\s+", ""); // Regular expression, too!
```

The elegant and (implicitly) recursive

```
filter (/=' ') x
```

in Haskell is less elegant in Java:

```
x.chars()
 .filter(ch ->ch!=' ')
 .collect (StringBuilder::new,
           StringBuilder::appendCodePoint,
           StringBuilder::append)
 .tostring();
```

Java strings and method references are an advanced topic.

More Recursive Functions

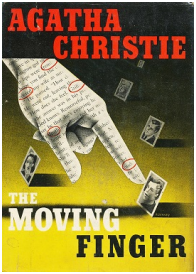
```
isPal []  = True
isPal [_] = True
isPal (x:rest) = x==last(rest)&&isPal(init(rest))
```

```
boolean isPalindrome (final String s) {
   final int len = s.length ();
   if (len<2) {
      return true;
   } else  {
      // Recurse to the center!
      return s.charAt(0)==s.charAt(len-1) &&
         isPalindrome (s.substring(1,len-1));
   }
}
```

More Recursive Functions

```
lex _ [] = False
lex [] _ = True
lex (a:as) (b:bs) = a<b || (a==b && lex as bs)
```

```
public static boolean lexicographic(String x,String y) {
  if (y.isEmpty()) return false;
  else if (x.isEmpty()) return true;
  else if (x.charAt(0) < y.charAt(0)) return true;
  else if (x.charAt(0) > y.charAt(0)) return false;
  else return lexicographic (
     x.substring(1), y.substring(1)));
}
```



Agatha Christie's book takes its name from quatrain 51 of Edward FitzGerald's translation of the Rubáiyát of Omar Khayyám:

> *The Moving Finger writes; and, having writ,*
> *Moves on: not all thy Piety nor Wit*
> *Shall lure it back to cancel half a Line,*
> *Nor all thy Tears wash out a Word of it.*

The poem, in turn, refers to Belshazzar's feast as related in the Book of Daniel, where the expression *the writing on the wall* originated.

More Recursive Functions

```
int largest (int[] a) {
   return largest (a, 0, Integer.MIN_VALUE);
}

int largest (int[] a, int start, int max) {
   if (start == a.length) {
      return max;
   } else {
      final int l = a[start]>max?a[start]:max;
      return largest(a, start + 1, l);
   }
}
```

More Recursive Functions

```
double sin (final double x) {
  if (Math.abs (x) < 0.005) {
    return x - x*x*x/6.0;  // An approximation for small x
  } else {
    return 2.0 * sin(x/2.0) * cos(x/2.0);
  }
}

double cos (final double x) {
  if (Math.abs (x) < 0.005) {
    return 1.0 - x*x/2.0;  // An approximation for small x
  } else {
    return 1.0 - 2.0*sin(x/2.0)*sin(x/2.0);
  }
}
```

Factorial

```
public class Main
   public static int fact (final int n) {
     if (n==0) return 1;
     else return n * fact(n-1);
   }
   public static void main (final String[] args) {
     System.out.println (fact(4));
   }
}
```

```
fact (0) = 1
fact (n) = n * fact(n-1)
```

```
fact 4 =
       = 4 * fact(3)
       = 4 * 3 * fact(2)
       = 4 * 3 * 2 * fact(1)
       = 4 * 3 * 2 * 1 * fact(0)
       = 4 * 3 * 2 * 1 * 1
       = 4 * 3 * 2 * 1
       = 4 * 3 * 2
       = 4 * 6
       = 24
```

The “left-over” work requires a lot of memory which is unfortunate (and unnecessary).

GCD

Compare with gcd.
gcd is tail recursive
the last action the function does is call itself.

GCD

```
public class Main
   public static int gcd (final int p, final int q) {
     if (q==0) return p;
     else return gcd (q, p%q);
   }
   public static void main (final String[] args) {
     System.out.println (gcd(1272, 216));
   }
}
```

```
gcd (p, 0) = p
gcd (p, q) = gcd (q, p%q)
```

```
gcd (1272, 216) =
                = gcd (216, 192)
                = gcd (192, 24)
                = gcd (24, 0)
                = 24
```

One can make the `fact` function tail recursive:

```
// Compute n!*r
public static int fact (final int n, final int r) {
    if (n==0) return r;
    else return fact(n-1, n*r);
}
```

This has the tremendous advantage of being tail recursive (the recursive call is the last thing the function does). Such a recursive function can be translated in a loop by the compiler avoiding the overhead of a procedure call to store the left-over work.

```
fact (4,1) =
      = fact(3,4)
      = fact(2,12)
      = fact(1,24)
      = fact(0,24)
      = 24
```

Exponential (Java)

Generalize and compute $y \cdot x^n$

```
pow(y,x,n)
    | n==0       = y
    | n==1       = x*y
    | even n     = pow(y,x*x,n`div`2)
    | otherwise = pow(x*y,x*x,(n-1)`div`2)
```

```
int pow (int y, int x, int n) {
   if (n == 0) return  y;
   else if (n == 1) return  x * y;
   else if (n%2==0) return pow(y, x*x,  n/2);
   else /* odd */   return pow(x*y,x*x,(n-1)/2);
}
```

Tail recursive!

Look at the tree of recursive calls made in computing fibonnaci.
Can one make fibonnaci tail recursive?
One can, if one generalizes the problem and introduces additional parameters.

```
fib3 (0,_,b) = b
fib3 (n,a,b) = fib3 (n-1, a+b, a)
```

```
fib(0)=fib3(0,1,0)=0
fib(1)=fib3(1,1,0)=fib3(0,1,1)=1
fib(2)=fib3(2,1,0)=fib3(1,1,1)=fib3(0,2,1)=1
fib(3)=fib3(3,1,0)=fib3(2,1,1)=fib3(1,2,1)=fib3(0,3,2)=2
fib(4)=fib3(4,1,0)=fib3(3,1,1)=fib3(2,2,1)=fib3(1,3,2)=fib3(0
```

Let us look at it more symbolically. Given the k and $k+1$th Fibonacci number compute the $n+k$th Fibonacci number.

```
fib3 (0,_,b) = b
fib3 (n,a,b) = fib3 (n-1,a+b,a)
```

and

```
fib (n) = fib3 (n,1,0)
```

So,

```
fib (n) = fib3 (n,1,0) = fib3 (n, fib(1), fib(0))
 = fib3 (n-1, fib(2), fib(1))
 ...
 = fib3 (n-k, fib(k+1), fib(k))
 ...
 = fib3 (0, fib(n+1), fib(n)) = fib(n)
```

We know `fib(k+2)= fib(k+1)+fib(k)`. To prove correctness of `fib3(n,a,b)`, we make the following equation hold for all n, k:

```
fib3 (n, fib(k+1), fib(k)) = fib (n+k)
```

For $n = 0$, if:

```
fib3 (0, a, b) = b
```

then the following holds:

```
fib3 (0, fib(k+1), fib(k)) = fib (0+k) = fib (k)
```

For $n > 0$, if $a = fib(k+1)$ and $b = fib(k)$ and

```
fib3 (n, a, b) = fib3 (n-1, a+b, a)
fib3 (n, fib(k+1), fib(k)) = fib (n+k)
```

and then the following holds

```
fib3 (n+1, fib(k+1)+fib(k), fib(k)) =
   fib3 (n, fib(k+2),fib(k+1)) = fib(n+k+1) = fib(n+1+k)
```

Recursive Definitions of Functions

The number of committees of r members from a total of n people:

$$C(n,r) = \begin{cases} 1 & \text{if } r = 0 \text{ or } r = n, \\ C(n-1,r-1) + C(n-1,r) & \text{otherwise} \end{cases}$$

More generally we allow $r = 0$ and $n = 0$ and define the number of *combinations* as the number of ways a subset of $r \leq n$ items can be chosen out of a set of $n \geq 0$ items. The function is sometimes known as the *choose function* and the value as the *binomial coefficient*.

$$C(n,r) = C_r^n = {}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Arranging the binomial coefficients in a triangle, gives us Pascal's famous triangle:

n/r	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Binomial Coefficient

The formula

$$C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

is interesting, but very bad for computation as the factorial grows very large even if the answer is small.

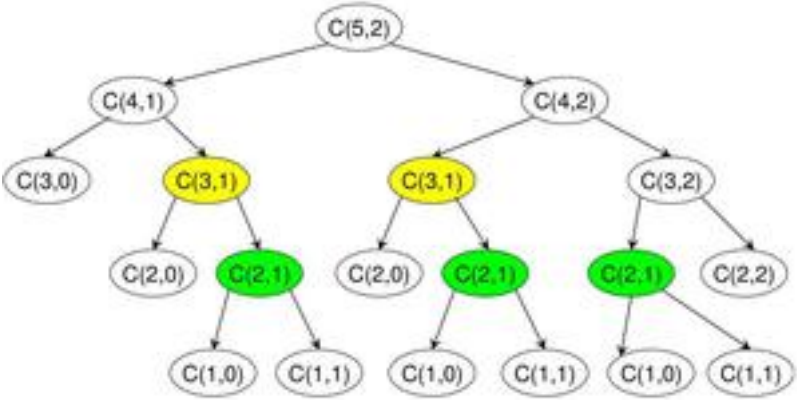
Binomial Coefficient

The following does not have large intermediate calculations, but has lots of recomputations.

$$C(n,r) = \begin{cases} 1 & \text{if } r = 0 \text{ or } r = n, \\ C(n-1,r) + C(n-1,r-1) & \text{otherwise} \end{cases}$$

```
// Compute: n choose r
int choose (final int n, final int r) {
   if (r==0 || r==n) return 1;
   else return choose(n-1,r) + choose(n-1,r-1);
}
```

Binomial Coefficient



Binomial Coefficient

Mathematical identities about the binomial coefficient.

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

$$\binom{n}{r} = \binom{n}{n-r}$$

$$\binom{n}{r} = \frac{n!}{(n-r)!r!}$$

$$\binom{n}{r} = \frac{n}{r}\binom{n-1}{r-1}$$

Sometimes the easiest way to conceive it is not the best way to program it. Here is an equivalent version that is more efficient.

$$C(n,r) = \begin{cases} 1 & \text{if } r = 0 \text{ or } r = n, \\ \frac{n}{rC(n-1,r-1)} & \text{otherwise} \end{cases}$$

```
choose (n,r)
  | r==0 || r==n = 1
  | otherwise    = n * choose (n-1,r) / r
```

```
// Compute: n choose r.  Assuming 0<=r<=n.
int choose (final int n, final int r) {
   if (r==0 || r==n) return 1;
   else return n * choose(n-1, r-1) / r;
}
```

Tail recursive or not?

Tail recursive or not? Not tail recursive.

Sometimes solving a more general problem is more efficient.

$$C(n, r, a) = \begin{cases} a & \text{if } r = 0 \text{ or } n = r, \\ C(n-1, r-1, a * n/r) & \text{otherwise} \end{cases}$$

```
choose (n,r,a)
  | r==0 || r==n = a
  | otherwise    = choose (n-1,r-1,a*n`div`r)
```

```
// Compute: a * (n choose r)
int choose (int n, int r, int a) {
   if (r==0 || r==n) return a;
   else return choose(n-1, r-1, a*n/r)
}
```

Generalizing to three arguments must be done carefully. The following does not work on integers because the division is done in the wrong order (at time when the accumulated value may not be divisible by r).

```
// Compute: n choose r times acc
int choose (int n, int r, int acc) {
   if (r==0 || r==n) return acc;
   else if (r==1) return acc*n;
   else return choose(n-1, r-1, acc*n/r)
}
```

$$C(n,r) = \frac{n}{r} \cdot \frac{n-1}{r-1} \cdot \frac{n-2}{r-2} \cdots \frac{n-k+1}{1}$$

$$C(n,r) = \frac{n}{r} \cdot \frac{n-1}{r-1} \cdot \frac{n-2}{r-2} \cdots \frac{n-k+1}{1}$$

Reversing the order helps.

$$C(n,r) = \frac{n}{1} \cdots \frac{n-k-1}{r-2} \cdot \frac{n-k}{r-1} \cdot \frac{n-k+1}{r}$$

```
choose (n,r,i,a)
  | i>=r      =a
  | otherwise=choose (n,r,i+1,a*(n-i)`div`(i+1))
```

```
// Compute: n choose r times acc
int choose (int n, int r, int i, int acc) {
  if (i>=r) return acc;
  else return choose (n,r,i+1,acc*(n-i)/(i+1))
}
```

```
// Compute: n choose r times acc
int choose (int n, int r, int i, int acc) {
  if (i>=r) return acc;
  else return choose (n,r,i+1,Math.multiplyExact(acc,(n-i))/(
```

```
int choose (int n, int r, int i, int a) {
   start:
   if (i>=r) return a;
   } else {
      i++; a = a*(n-1)/i; goto start;
   }
}
```

An example of the tail recursive choose function.

```
choose (5, 3, 1) =
               = choose (4, 2, 1*5/3)
               = choose (3, 1, 1*5/3  * 4/2)
               = 3 * 1*5/3 * 4/2
               = 5 * 2
               = 10

choose (5, 3, 0, 1) =
               = choose (5, 3, 1, 1*5/1)
               = choose (5, 3, 2, 1*5/1 * 4/2)
               = 10
```

Recursive Definitions of Functions

The number of ways to form k (non-empty) teams from a total of n people (everybody is on one of the teams):

$$s(n,k) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = k, \\ 1 & \text{if } k = 1, \\ s(n-1,k-1) + k \cdot s(n-1,k) & \text{otherwise} \end{cases}$$

To understand the recurrence, observe that a person "n" is on a team by himself or he is not. The number of ways that "n" is a team by himself is $s(n-1,k-1)$ since we must partition the remaining $n-1$ people in the the available $k-1$ teams. The number of ways that "n" is a member of one of the k teams containing other people is given by $k \cdot s(n-1,k)$.

Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$

n/k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	3	1				
3	1	7	6	1			
4	1	15	25	10	1		
5	1	31	90	65	15	1	
6	1	63	301	350	140	21	1

Stirling numbers of the second kind (sequence A008277 in OEIS):

$$1, 1, 1, 1, 3, 1, 1, 7, 6, 1, 1, 15, 25, 10, 1, \ldots$$

Recursive Definitions of Functions

The number of expressions containing n pairs of parentheses which are correctly matched.
For $n = 3$ we have five: ((())), ()(()), ()()(), (())(), (()())

$$C_n = \begin{cases} 1 & \text{if } n = 0 \\ \frac{2(2n-1)}{n+1} C_{n-1} & \text{otherwise} \end{cases}$$

The first Catalan numbers (sequence A000108 in OEIS) are

$$1, 1, 2, 5, 14, 42, 132, \ldots$$

OEIS A001006

Delta Wave from the ICPC 2010 Asia Regionals.

OEIS A001006

Can one make Motzkin tail recursive?
Motzkin numbers: number of ways of drawing any number of non-intersecting chords joining n (labeled) points on a circle.

$$1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188,$$

```
f n a b = (2*n+1)*a)  +  (3*n-3)*b) `div` (n+2)

motzkin 0 = 1
motzkin 1 = 1
motkzin n = f (motzkin(n-1)) (motzkin(n-2))

motzkin2 :: Integer -> (Integer,Integer)
motzkin2 0 = (1, 1)
motzkin2 1 = (1, 2)
motzkin2 n = (f n a b, a)
   where (a,b) = motzkin2 (n-1)
```

The trick is to again introduce additional parameters and make the following equation hold:

```
motzkin3 (n, motzkin(k+1), motzkin(k)) =
   motzkin (n+k)
```

Given the k and $k+1$th Motzkin number compute the $n+k$th Motzkin number.

```
motzkin3 (0,a,b) = b
motzkin3 (1,a,b) = a
motzkin3 (n,a,b) = motzkin3 (n-1, f n a b, a)
```

Then

```
motzkin n = motkzin3 (n,1,1)
```

```
bin 0 = []
bin n = let b=bin n-1 in map ('0'+) b + map ('1'+ b)
```

```
void printBin (String prefix, int digits) {
  if (digits==0) println (prefix);
  else {
     printBin ('0'+prefix, digits-1);
     printBin ('1'+prefix, digits-1);
  }
}
```

Recursion is always easy to understand

Why? No time is involved.

Recursion is easy to understand and easy to do. It possible to very powerful things with recursion. In this way it is surprisingly easy to exceed the ability of the computer.

- stack overflow
- excessive recomputation

You can do anything with recursion.

```
int add (int n, int m) {
  if (m==0) return n;
  else add (n, m-1);
}
```

It is easier to make a correct program more efficient than to make a buggy program more correct.

A program that does what you think it does is much better than a program that might do what you want it do.