Static Methods

Anatomy of a static method, S&W, 2.1, figure page 188.

Class.name or just “name”

Scope, S&W, 2.1, figure page 189.

Overloading

```
public static int abs (final int x) {
   return x<0?-x:x;
}
public static double abs (final double x) {
   return x<0.0?-x:x;
}
```

Overload resolution does take the return type into consideration. Java does promote int values to double, but try to avoid taking advantage of that as that will force the reader to learn the rules of overload resolution.

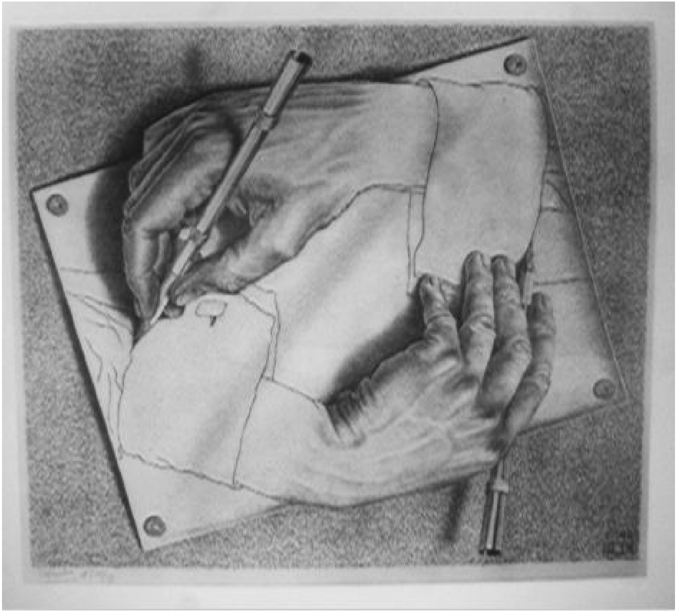
Libraries

A class can contain (static) code which can be used by another class.

For example, one might use `Math.hypot`.

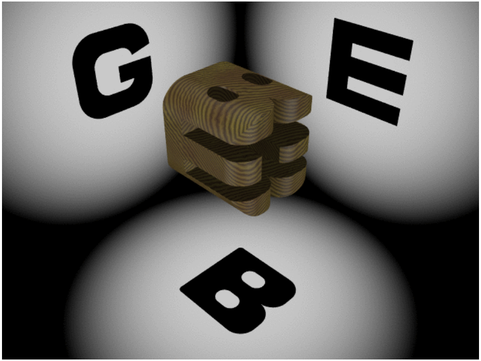
See page 219, in S&W.

Recursion



Recursion

Pulitzer Prize winning book on recursion
Gödel, Esher, Bach: An Eternal Golden Braid by Douglas R. Hofstader.



Recursion

> *Anything that, in happening, causes itself to happen again, happens again.*
>
> Douglas Adams, *Mostly Harmless*

Recursion

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.

Recursion

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

- Linked data structures
- Mergesort, quicksort, greatest common divison, fast fourier transform
- The file system where a folder constains files and other folders.

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 expression the complet meaning of a loop.

Reasoning about iteration is difficult because it requires actions over time.

Recursion

To understand and create solutions to complex problems it is necessary to decomposed them into smaller problems, The solutions to small problems are composed into the solution of the original problems. Subprograms are an indispensable part in writing programs to solve problems.

In some cases, the smaller problem is fundamentally the same as the original problem. A recursive solution has been found.

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.

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 all.)

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.

Recursive Definitions of Functions in Mathematics

Lots of recursive definitions are found in mathematics.

A recursive definition is very easy to translate into Java.

(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

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

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

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

For $x \geq y$,

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

Recursive Definitions of Functions

For $0 \leq r \leq n$,

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

For $0 \leq k \leq n$,

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

For $0 \leq k \leq n$,

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

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 n>=0;
   if (n==0) return 1;
   else return n * fact(n-1);
}

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

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

Alternate Function Definitions

One might accept the clumsy:

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

which is clearly realized in the Java code:

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

Exponential

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

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

Fibonacci

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

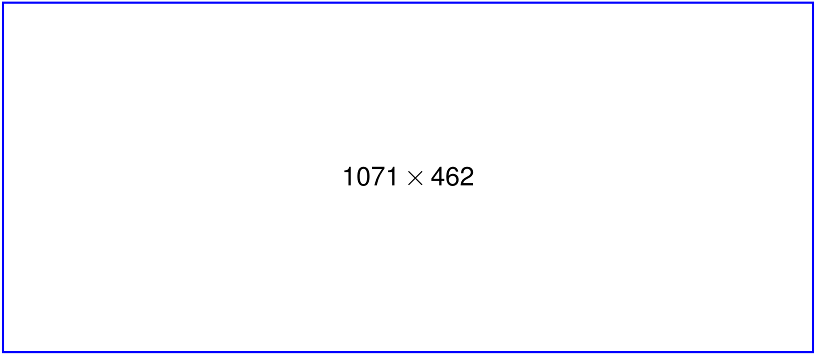
```
int fib (final int n) {
  if (n==1) return 1;
  else if (n==2) return 1;
  else return fib(n-1) + fib(n-2)
}
```

GCD

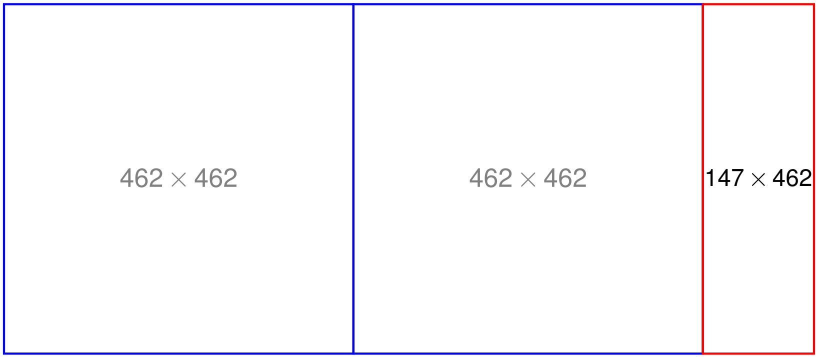
```
int gcd (final int p, final int q) {
  if (q==0) return p;
  else return gcd (q, p%q);
}
```

One way of visualizing the recusrive 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 p and q.

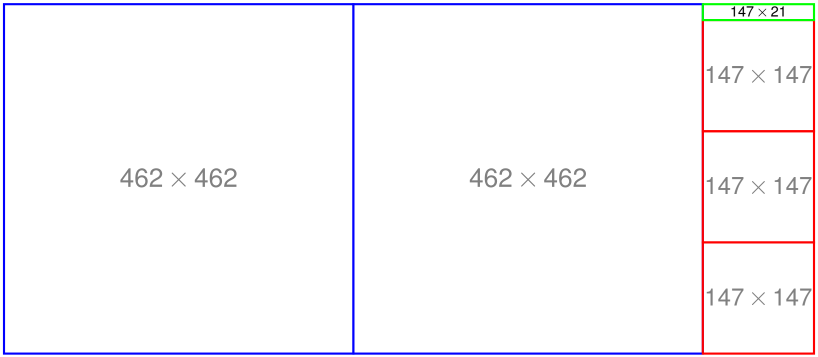
GCD of 1071 and 462 is 21



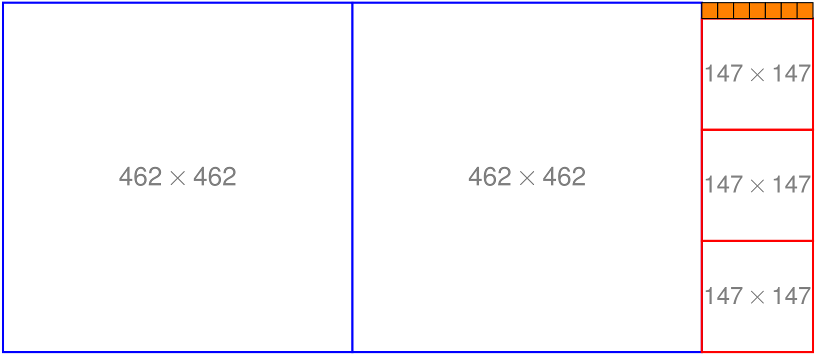
GCD of 1071 and 462 is 21



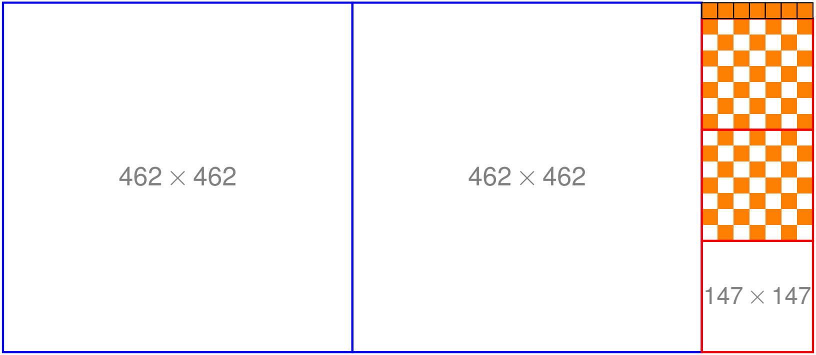
GCD of 1071 and 462 is 21



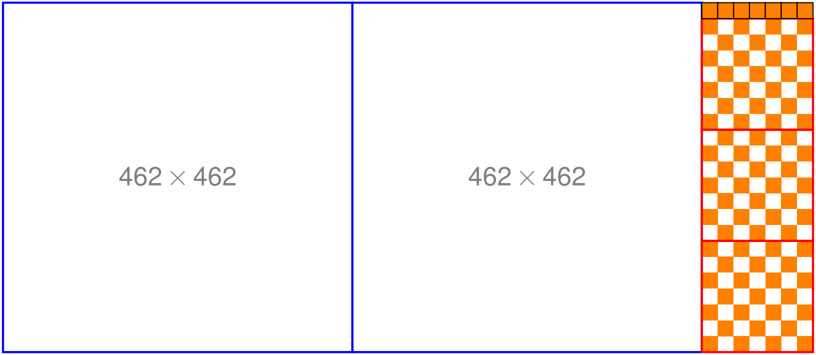
GCD of 1071 and 462 is 21



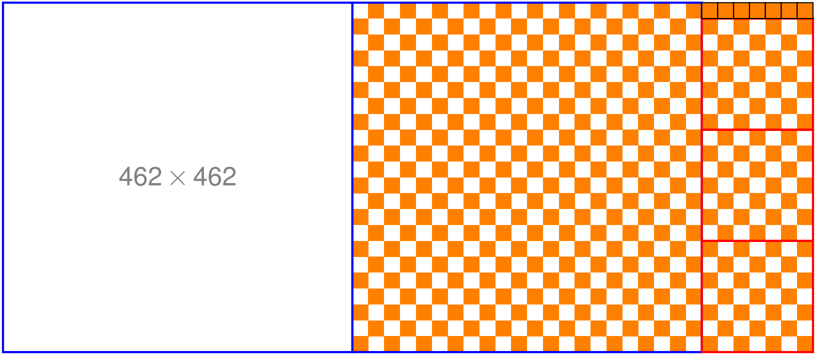
GCD of 1071 and 462 is 21



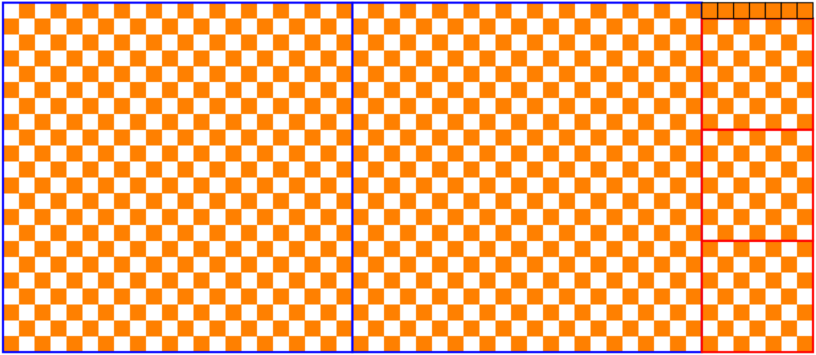
GCD of 1071 and 462 is 21



GCD of 1071 and 462 is 21



GCD of 1071 and 462 is 21



Binomial coefficient

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,0)= 1
c(n,r)= if (n==r) then 1 else 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 $0 \leq k \leq n$,

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

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

Stirling Numbers of the Second Kind

For $0 \leq k \leq n$,

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

```
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.

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

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

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

More Recursive Functions

```
public static boolean lexicographic(String x,String
  if (y.length()==0) return false;
  else if (x.length()==0) 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)));
}
```

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
  } 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
  } 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
```

Look at the tree of recursive calls made in computing fibonnaci.
Can one make fibonnaci tail recursive?
The trick is to again introduce additional parameters and make the following equation hold:

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

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

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

Then

```
fib3 (n, fib(k+1), fib(k)) = fib3 (n-1, {fib(k+2)=}
motzkin (n, motzkin (k+1), motzkin k) = motzkin (n-
fib3 (n, fib(k+1), fib(k)) = fib (n+k)
fib (n) = fib3 (n,1,0) = fib3 (n, fib(1), fib(0))
```

Can one make Motzkin tail recursive?

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

motzkin 0 = 1
motzkin 1 = 1
motkzin n = f  (m(n-1)) (m(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
```

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

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

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)
```

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*.

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

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.

The Java program has lots of recomputation:

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

Binomial Coefficient

Mathematical identities about the binomial coefficient.

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

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

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

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

Binomial Coefficient

Mathematical identities about the binomial coefficient.

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

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

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

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

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

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

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

Sometimes the solving a more general problem presents a solution which is more efficient.

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

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

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

Reversing the order helps.

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

```
// Compute: n choose r times acc
public static 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)
}
```

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

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

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);
}
```

```
int fib (int n) {
  if (n==0) return 0;
  else if (n==1) return 1;
  else return fib(n-1)+fib(n-2)
}
```

Makes a simple and easy definition, but hidden in the computation is the wasteful recomputation of my common values.

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

int fib3 (int n, int a, int b) {
  if (n==0) return a;
  else fib3 (n-1, a+b, a)
}

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

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.