

Recursive

A question

- People often give out their telephone number as a word representing the seven-digit number. For example, if a number is 866-2665, I could tell people my number was “TOOCOOL” instead of the hard-to-remember seven-digit number.
- Not that many other possibilities (most of which are nonsensical) can represent 866-2665. You can see how letters correspond to numbers on a telephone keypad.
- Write a function that takes a seven-digit telephone number and prints out all of the possible “words” or combinations of letters that can represent the given number. Because the 0 and 1 keys have no letters on them you should change only the digits 2-9 to letters.

1	ABC 2	DEF 3
GHI 4	JKL 5	MNO 6
PRS 7	TUV 8	WXY 9
*	0	#

Objectives

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for using arrays and linked lists
- To understand how to use recursion to calculate permutation
- To understand how to use recursion to solve the Towers of Hanoi problem

Iteration

- Iteration
 - Loops: for, while

Recursion

- **Recursion** is a problem-solving approach in which a problem is solved using repeatedly applying the **same** solution to **smaller** instances.
 - Each instance to the problem has size.
 - An instance of size n can be solved by putting together solutions of instances of size at most $n-1$.
 - An instance of size 1 or 0 can be solved very easily.

Introduction

- Iteration
 - Loops: for, while
- Recursion
 - A function refers to itself in its own definition
 - Function call: f1 calls f2
 - Function call: f1 calls f1?
 - Provides an **elegant and powerful alternative** for performing **repetitive** tasks

Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Other examples
- Towers of Hanoi, Complexity
- Reasoning

Several questions

- Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

$$\mathit{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \mathit{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

One or more base cases

One or more recursive cases

Implementation

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

Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Other examples
- Towers of Hanoi, Complexity
- Reasoning

Greatest Common Divisor

- The **greatest common divisor**, sometimes also called the **highest common divisor**, of two positive integers a and b , denoted as $\text{gcd}(a,b)$, is the largest divisor common to a and b .
- The greatest common divisor g is the largest natural number that divides both a and b without leaving a remainder.
 - E.g., $\text{gcd}(3,8)=1$, $\text{gcd}(4,16)=4$
- Brute forth algorithm

```
public static int gcdl (int a, int b){  
    int n = (a<b)?a:b;  
    int gcd = 1, i = 1;  
    while (i <= n) {  
        if (a % i == 0 && b % i == 0) gcd = i;  
        i++;  
    }  
    return gcd;  
}
```

Euclidean algorithm (Euclid's algorithm) $\gcd(a,b)$

- It is named after the ancient Greek mathematician Euclid.
- If $a=0$, $\gcd(a,b)=b$
- If $b=0$, $\gcd(a,b) = a$
- If $a \neq 0$ and $b \neq 0$, $\gcd(a,b)=\gcd(b,a \% b)$

Lemma

- **Lemma.** Suppose $d, u, v \in \mathbb{Z}$. Then $d \mid u$ and $d \mid v$ if and only if $d \mid v$ and $d \mid u - qv$ where $q \in \mathbb{Z}$.
- Proof:
- (\Rightarrow) If $d \mid u$ and $d \mid v$, then clearly $d \mid v$. Since $d \mid u$, there exists $n \in \mathbb{Z}$ such that $u = nd$. Similarly, there is $m \in \mathbb{Z}$ such that $v = md$. Then $u - qv = nd - qmd = (n - qm)d$, so $d \mid u - qv$ as well.
- (\Leftarrow) Again, it is trivial to see that if $d \mid v$ and $d \mid u - qv$, then $d \mid v$. We can write $v = md$ and $u - qv = nd$ for some $m, n \in \mathbb{Z}$. Then $u = nd + qv = nd + qmd = (n + qm)d$, so $d \mid u$.
- Notation $d \mid u$: u is divisible by d , e.g., 15 is divisible by 3.

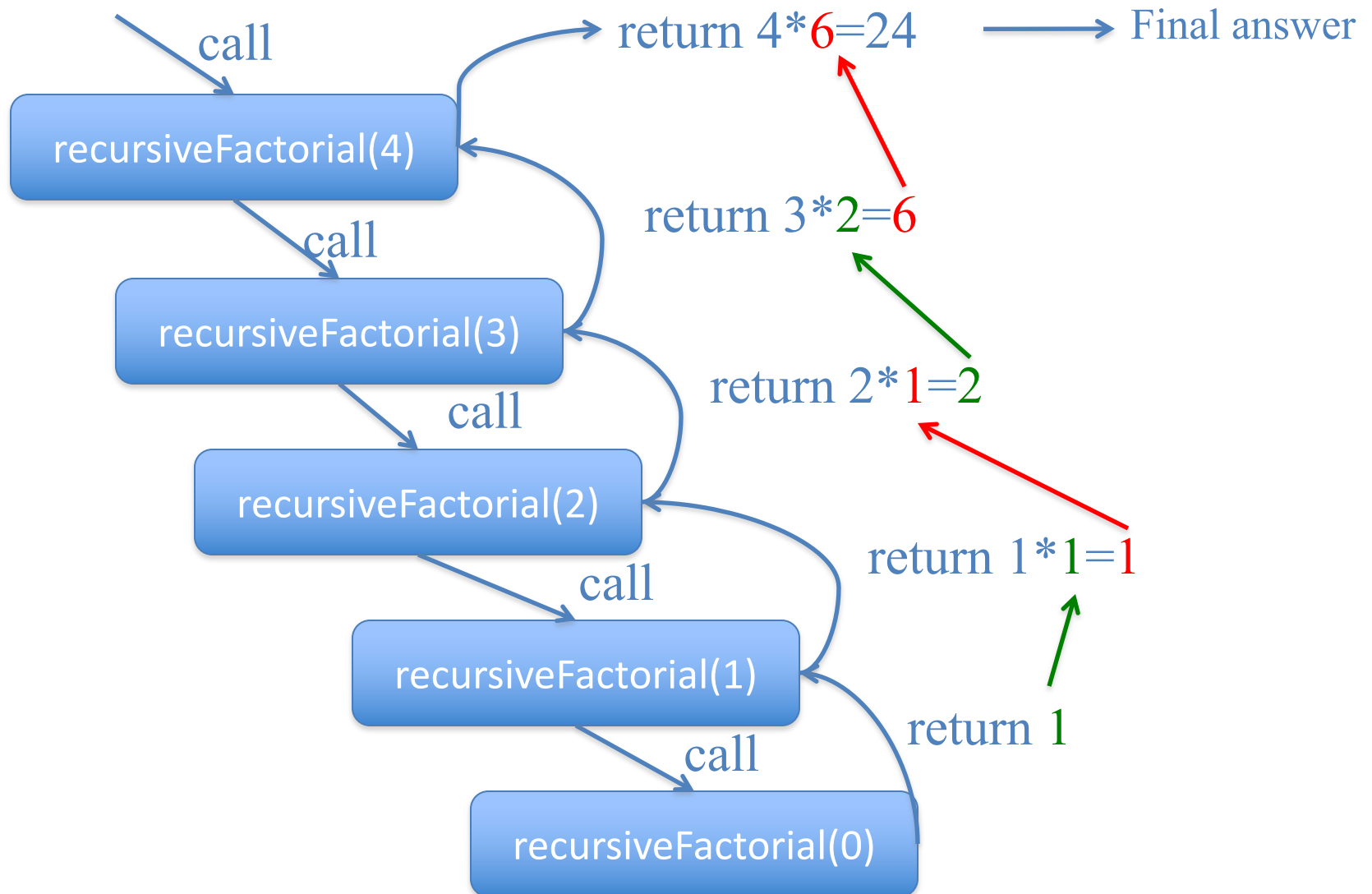
Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Other examples
- Towers of Hanoi, Complexity
- Reasoning

Recursion trace

- Illustrate the execution of a recursive function definition by means of **recursion trace**.
 - Each entry of the trace corresponds to a recursive call.
 - Each new recursive function call is indicated by an **arrow** to the newly called function.
 - When the function **returns**, an arrow showing this return is drawn. The return value may be indicated with this arrow.

A recursive trace for the call recursiveFactorial (4)



System Processing of a Recursion

- Push onto a **stack** the information of the current execution
- Execute the recursive call
- Retrieve the information from the stack by pop
- Too many recursive calls without pop will result in stack overflow.

Recursion versus Iteration

- Recursive methods are often **more expensive** than iterative methods because the stack overhead is larger than the loop overhead
- Recursive methods are **easier to write and conceptualize.**

Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Other examples
- Towers of Hanoi, Complexity
- Permutation
- Reasoning

Linear recursion

- A function is defined so that it makes **at most one recursive call** each time it is invoked. (Simplest form of recursion)
- **SumLinearRecursive(A,n)**

```
public static int SumLinearRecursive(int[] A,int n)
{
    if(n==1) return A[0];
    else return (SumLinearRecursive(A,n-1)+A[n-1]);
}
```
- **Recursion trace** for SumLinearRecursive(A,5)
- n calls for an array of size n , **time** proportional to n , **space** proportional to n

Tail recursion

- An algorithm uses **tail recursion** if
 - it uses **linear recursion** and
 - it makes a **recursive call as its very last** operation
- **SumLinearRecursive** does not use tail recursion

```
public static int SumLinearRecursive(int[] A,int n)
{
    if(n==1) return A[0];
    else return (SumLinearRecursive(A,n-1)+A[n-1]);
}
```

Tail recursion

- `ReverseArrayRecursive` uses tail recursion
call `ReverseArrayRecursive(A,0,A.length-1);`
- Method implementation

```
public static void ReverseArrayRecursive(int[] A, int i,int j){  
    if(i<j){  
        int tmp = A[i];  
        A[i]=A[j];  
        A[j]=tmp;  
        ReverseArrayRecursive(A,i+1,j-1);  
    }  
}
```

Tail recursion

- An algorithm uses **tail recursion** if
 - it uses **linear recursion** and
 - it makes a **recursive call as its very last** operation
- **ReverseArrayRecursive** uses tail recursion
- When an algorithm uses tail recursion, we **convert the recursive algorithm into a non-recursive** one, by iterating through the recursive calls rather than calling them explicitly.

```
public static void ReverseArray (int[] A, int i,int j){  
    while i<j do{  
        swap A[i] and A[j]  
        i=i+1  
        j=j-1  
    }  
}
```

Binary Recursion

- An algorithm makes **two recursive calls**
- Solve **two similar halves** of some problem
- **SumBinaryRecursive(A,i,n)**
- **Recursion trace**
- The **maximum number of function instances** that are active at the same time: $1 + \log_2 n$
 - Space (improves SumLinearRecursive)
 - Time: proportional to n ($2n-1$ total function calls)

Multiple Recursion

- A function can make m recursive calls where $m > 2$
- Very commonly used when we want to enumerate various configurations in order to solve a **combinatorial puzzle**
- **Permutation**

Permutation

$\{1,2,3\}$, permutations: $n!$

- $[1]\{2,3\}$
 - $[1,2]\{3\} \rightarrow [1,2,3]$
 - $[1,3]\{2\} \rightarrow [1,3,2]$
- $[2]\{1,3\}$
 - $[2,1]\{3\} \rightarrow [2,1,3]$
 - $[2,3]\{1\} \rightarrow [2,3,1]$
- $[3]\{1,2\}$
 - $[3,1]\{2\} \rightarrow [3,1,2]$
 - $[3,2]\{1\} \rightarrow [3,2,1]$

permutation

```
public static <E> void PermuteArray (E[] array, int prefixLen)
```

- Base case: prefixLen =array.length
- Recursive case: prefixlen <array.lenth
- For (each position from [prefixlen] to array.length)
 - swap array[i] and array[prefixLen]
 - permutate all the elements in array[prefixLen+1] ... array[array.length-1] by recursive call, **PermuteArray** (array,prefixLen+1);
 - swap back array[prefixLen] and array[i]

- Main function call

```
Integer[] A= new Integer []{1,2,3}
```

```
PermuteArray(A,0)
```

```
A={1, 2, 3 }
```

```
PermuteArray({1,2,3},0)
```

permutation

PermuteArray({1,2,3},0)

--swap A[0] and A[0], PermuteArray({1,2,3},1), swap back A[0] and A[0]

--swap A[0] and A[1], PermuteArray({2,1,3},1), swap back A[1] and A[0]

--swap A[0] and A[2], PermuteArray({3,2,1},1), swap back A[2] and A[0]

PermuteArray({1,2,3},1)

--swap A[1] and A[1], PermuteArray({1,2,3},2), swap back A[1] and A[1]

--swap A[1] and A[2], PermuteArray({1,3,2},1), swap back A[1] and A[2]

PermuteArray({1,2,3},2)

--swap A[2] and A[2], PermuteArray({1,2,3},3), swap back A[2] and A[2]

PermuteArray({1,2,3},2) print 1,2,3

Why using recursion?

- The subtask is simpler

Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- **Other examples**
- Towers of Hanoi, Complexity
- Reasoning

Solve a problem recursively

- **Trick**: assume that you know how to solve the problem on a smaller input
- **Solution**: break a problem to smaller problems
 - Figure out **what is/are the sub-problems** that come up in solving your problem
 - Figure out **how to compose the solution** to your original problem from the solution to the sub-problems
 - Provide a **base case**
- Correctness (termination, correctness)
- Efficiency: time/space complexity

Other examples

- **Search** a given value e from an array A whose values are sorted ascending
- Calculate the **number of digits** of a given number x
 - E.g., when $x=10$, the number of digits is 2
 - When $x=134$, the number of digits is 3
- Towers of **Hanoi**

Number of digits

```
int digits(int n)
{
    if(n<10)&&(n>-10) return 1;
    else return (1+digits(n/10));
}
```

Outline

- Factorial
- Greatest common divisor
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Other examples
- **Towers of Hanoi, Complexity**
- Reasoning

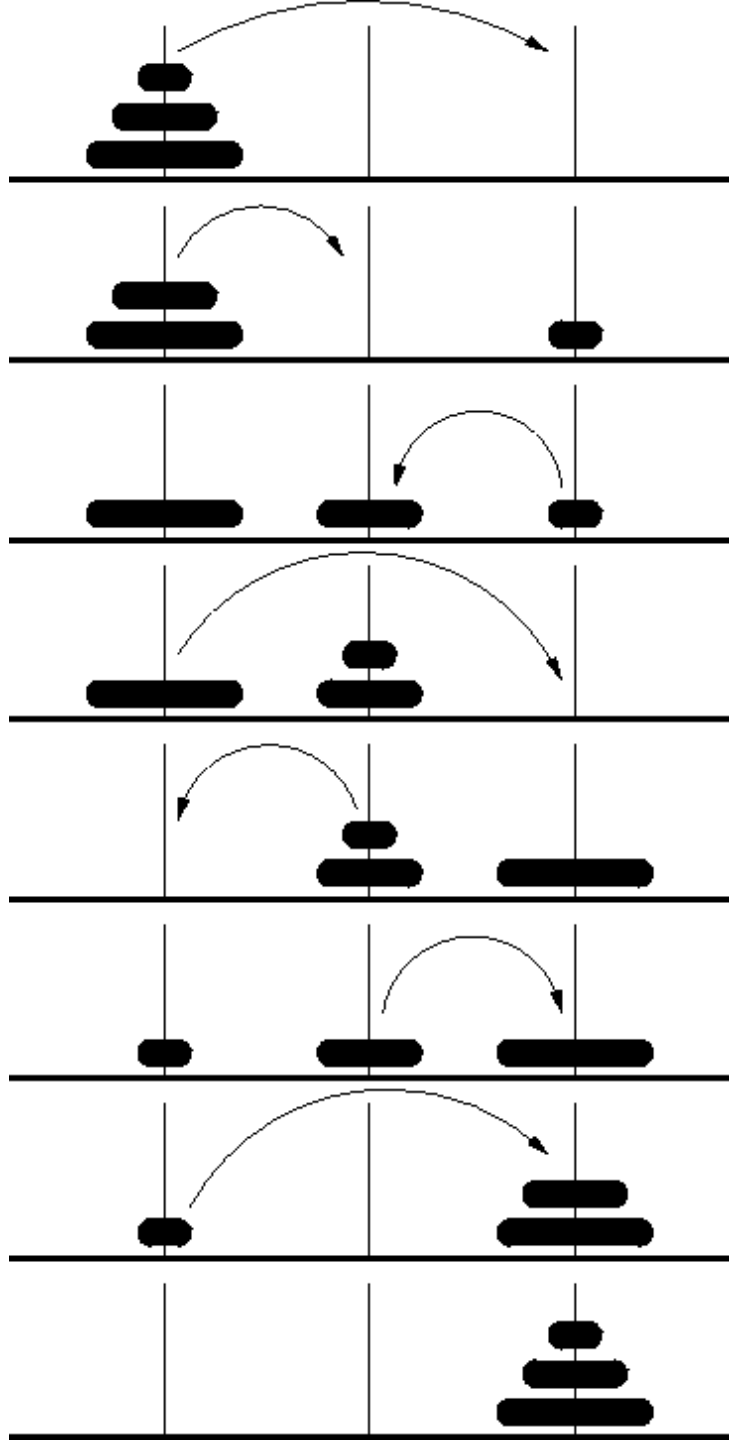
Towers of Hanoi

- The **Towers of Hanoi** puzzle was invented in the 1880s by Douard Lucas, a French mathematician.
- The puzzle consists of **three pegs** on which a set of **disks**, each with a different **diameter**, may be placed.
- **Initially** the disks are stacked on the leftmost peg, in order of size, with the largest disk on the bottom.

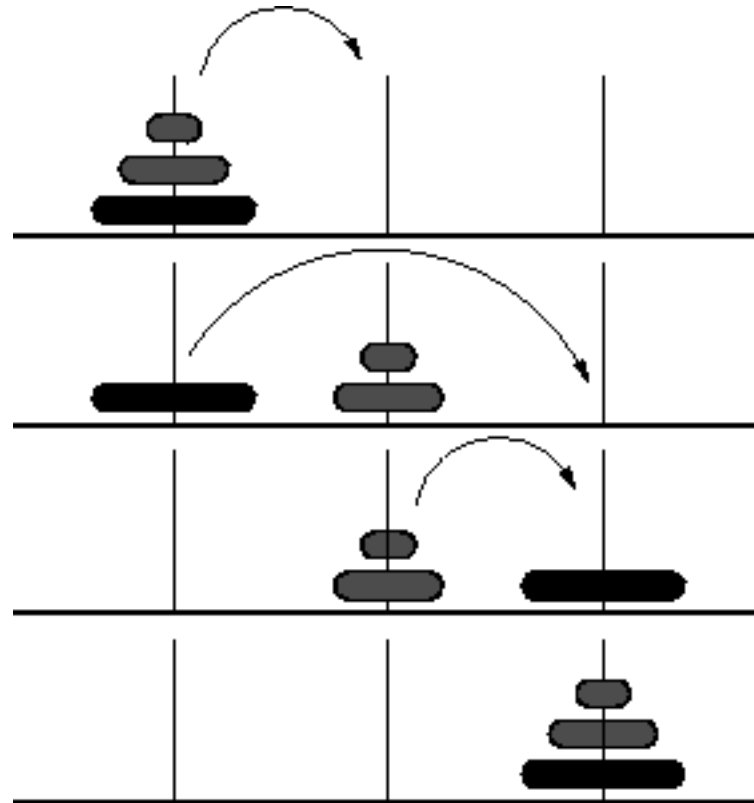


Goal

- The goal of the puzzle is to **move all** the disks from the **leftmost** peg to the **rightmost** peg, adhering to the following **rules**
 - Move only **one disk at a time**.
 - A **larger** disk may not be placed on top of a smaller disk.
 - All disks, except the one being moved, must be on a peg.



Strategy



Strategy

- The rules imply that **smaller disks** must be ``out of the way'' to move larger disks from one peg to another.
- General strategy for moving disks from the original peg to the destination peg:
 - Move the $N-1$ **topmost** disks from the original peg to the **extra** peg.
 - Move the **largest** disk from original peg to **destination** peg.
 - Move $N-1$ disks from the extra peg to the **destination** peg.

Recursive solution

- This strategy lends itself to a recursive solution.
- Step 1 and 3 are the **same problems** over and over again: move a stack of disks.
- The **base case** for this problem occurs when we want to move a ``stack'' that consists of only one disk. This step can be accomplished without recursion.

Outline

- Factorial
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Fractal
- Towers of Hanoi, **Complexity**
- Reasoning

Complexity

- Let the time required for moving n disks be $T(n)$.
- There are 2 recursive calls for $n-1$ disks and one constant time operation to move a disk from 'from' peg to 'to' peg . Let it be k_1 . Therefore,
 - $T(n) = 2 T(n-1) + k_1$
- Analysis
 - $T(1) = k_1$
 - $T(2) = 2 k_1 + k_1$
 - $T(3) = 4 k_1 + 2k_1 + k_1$
 - $T(4) = 8 k_1 + 4k_1 + 2k_1 + k_1$
 - $T(n) = (2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0)k_1 = (2^n - 1) k_1$

Space complexity

- $T(n) = T(n-1) + k$
- $T(1) = k$
- $T(2) = 2k$
- $T(n) = nk$
- $O(n)$

Outline

- Factorial
- Recursion trace
- Linear recursion, tail recursion, binary recursion, multiple recursion
- Fractal
- Towers of Hanoi, Complexity
- Permutation
- Reasoning

Reasoning about recursion

- Prove two facts
 - Termination
 - There is **no infinite recursion**
 - Correctness
 - Recursive method's **results are correct**

Termination

- Find a **variant expression** and a **threshold**
 1. Between recursive calls, the value of the variant expression decrease at least some **fixed amount**
 2. If the value of the **variant expression** \leq **threshold**, the method terminates (stopping condition)

Termination

- One-Level recursion
 - A **stopping** case
 - Or makes a **recursive call that is a stopping** case
- Multiple-level (deeper) recursion calls
 - **Variant expression**

Proof of correctness

- Recursive programming is related to mathematical induction
 - Induction: a technique of proving that some statement is true for n
- Proof of correctness is similar to Proof-by-induction
 - Proof by induction
 - Prove the statement is true for the base case (size 0, 1, or whatever)
 - Show that if the statement is assumed true for n , then it must be true for $n+1$
 - E.g., prove by induction $1 + 2 + 3 + \dots + n = n(n + 1)/2$

Proof of Correctness

- Recursive proof is similar to induction. Verify that:
 - The **base case** is recognized and solved correctly
 - Each **recursive case** makes progress towards the base case
 - If all **smaller problems** are solved correctly, then the original problem is also solved correctly.

Correctness - details

- Induction, in each call
 - Meet precondition and postcondition
 - Correct logic
- **Base step:** Whenever the method makes no recursive calls → it meets its precondition/postcondition
- **Induction step:** Whenever the method is activated and all the recursive calls meet their precondition/postcondition → the original call also meets its precondition and postcondition

Example

- Design a recursive method to compute the number of digits in an integer n . And prove (1) it can terminate, (2) it is correct.

```
int digits(int n)
{
    if(n<10)&&(n>-10) return 1;
    else return (1+digits(n/10));
}
```

Termination Proof - # of digits

- Proof of termination
 - Let us first define a variant expression. In this problem, we define a variant expression to be “the number of digits in n ”
 - Next we check how this variant expression changes. This variant expression decreases 1 in every recursive call.
 - The stopping case is the base case when the variant expression is one.

Proof of Correctness- # of digits

- Base case:
 - If n is in $(-10, 10)$, the number of digits is 1, which is correct.
- Induction step:
 - Induction hypothesis: suppose that $n/10$ has the correct number of digits, which is $x-1$.
 - Now we show that we can calculate the number of digits of n correctly.
 - Since $n/10$ reduces the number of digits by 1, # of digits of n is # of digits of $n/10 + 1$, which is the recursive code.

Towers of Hanoi

- Proof of Correctness
 - Base case: works correctly for moving one disk
 - Induction step:
 - Assume that it works correctly for moving $(n-1)$ disks
 - It follows that it works correctly for moving n disks.

gcd

```
int gcd(int a, int b) {  
    /*Pre: a>b; b≥0*/  
    /* Post: gcd(a, b) = GCD(a, b) */  
    if(b==0) return a;  
    else return gcd(b, a%b);  
}
```

Proof of correctness

- We prove its correctness by induction over N , the number of recursive calls.
- Base case: $b=0$, $N=0$, there for $\text{gcd}(a,b)=\text{gcd}(a,0)=a$. CORRECT.
- Inductive case:
 - Let a_n and b_n be the a and b in $\text{gcd}(a,b)$ after N recursive calls where $N>0$.
 - Let q and r be the quotient and the remainder of dividing a_n into b_n , so that $a_n = qb_n + r$ and $0 \leq r < b_n$ since $r = a_n \% b_n$.
 - $\text{gcd}(a_n, b_n) = \text{gcd}(b_n, a_n) = \text{gcd}(b_n, qb_n + r) = \text{gcd}(b_n, r)$
 - The next recursive call
 - $\text{gcd}(a_{n-1}, b_{n-1}) = \text{gcd}(b_n, r)$

Example

- $\text{Power}(\text{base}, \text{exp}) = \text{base}^{\text{exp}}$
- Rules:
 - If $\text{exp} > 0$, $\text{base}^{\text{exp}} = \text{base} * \text{base} * \text{base} * \dots$
 - If $\text{exp} = 0$, $\text{base}^0 = 1$
 - If $\text{exp} < 0$, $\text{base} \neq 0$, $\text{base}^{\text{exp}} = 1/\text{base}^{-\text{exp}}$
 - If $\text{exp} < 0$, $\text{base} = 0$, illegal

Summary

- Why do we need to have recursion?
- Recursive algorithm
 - Base cases
 - Recursive cases
- Recursion: linear, binary, multiple, tail recursive
- Algorithms
 - Factorial, randomFactal, Sum, Fibonacci, Tower of Hanoi, Permutation
 - Recursive algorithm design!!!!
- Reasoning
 - No infinite recursion
 - Correctness