GIUSEPPE TURINI

CS-102 COMPUTING AND ALGORITHMS 2
LESSON 01
RECURSION - THE MIRRORS

DATA ABSTRACTION AND PROBLEM SOLVING WITH JAVA, **CHAP 3, 137-195**.

HIGHLIGHTS

Recursive Solutions

- A Recursive Valued Method: The Factorial of an Integer
- The Box Trace
- A Recursive Void Method: Writing a String backward

Counting Things

- Multiplying Rabbits: The Fibonacci Sequence
- Organizing a Parade
- Mr. Spock's Dilemma: Choosing k out of n Things

Searching an Array

- Finding the Largest Item in an Array
- Binary Search
- Finding the k^{th} Smallest Item in an Array

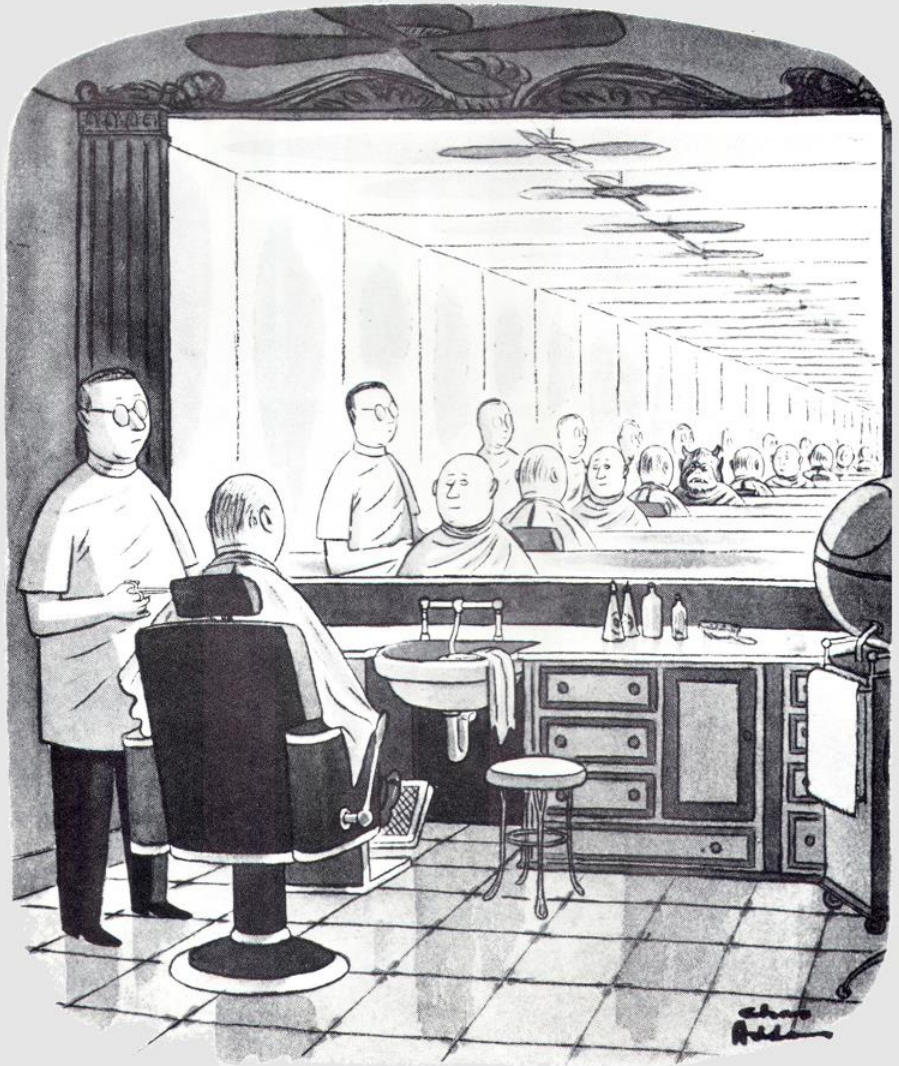
Organizing Data: The Towers of Hanoi

Recursion and Efficiency

THE MIRRORS CONCEPT

Recursion *breaks a problem into several smaller problems … of exactly the same type as the original problem –* ***mirror images****, so to speak.*

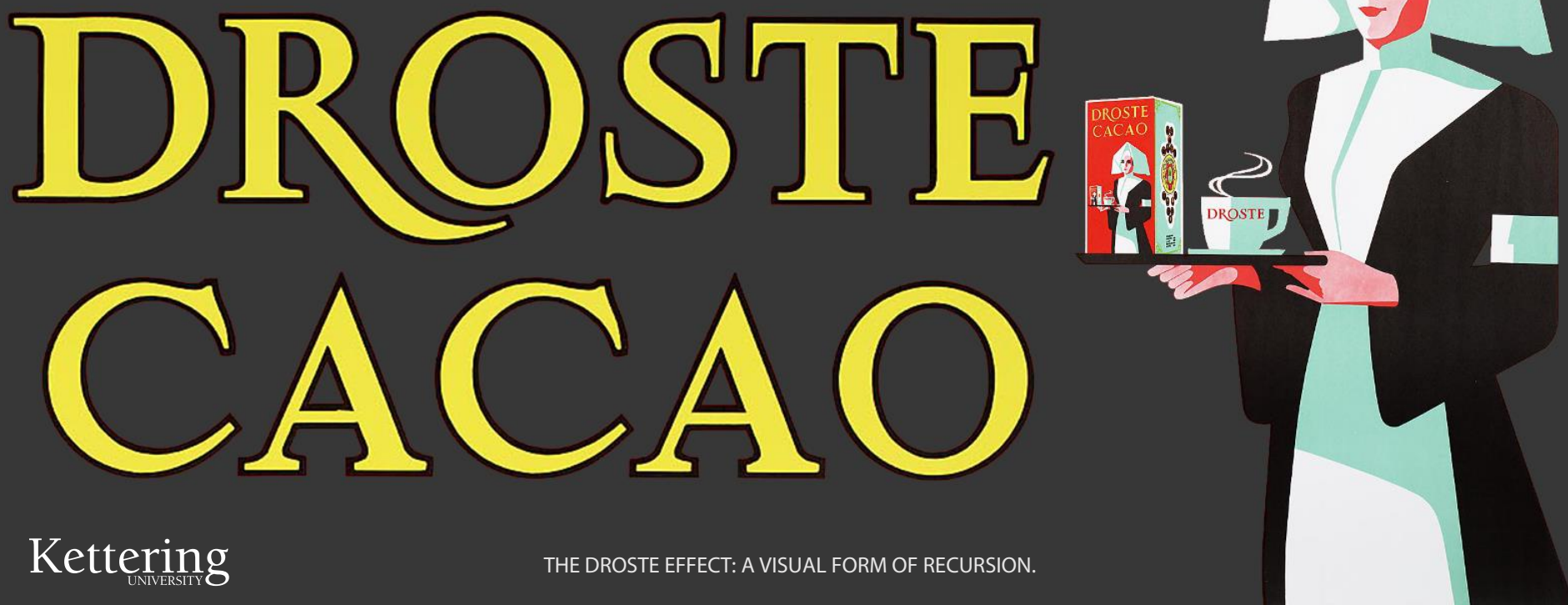
A ***recursive solution*** *solves a problem by solving a smaller instance of the same problem! Then, it solves this new problem by solving an even smaller instance of the same problem.*
*Eventually, the new problem will be so small that its solution is known (****base case****), and Will allow to solve the original problem.*



AN ILLUSTRATION BY CHARLES ADDAMS.

VISUAL RECURSION OR THE DROSTE EFFECT

The **Droste effect** (aka "mise en abyme" in art) is the effect of a picture appearing within itself, in a "similar" place. The appearance is recursive, and in theory this could go on forever.



Kettering UNIVERSITY

THE DROSTE EFFECT: A VISUAL FORM OF RECURSION.

RECURSIVE SOLUTIONS 1

RECURSION

An extremely powerful problem-solving technique.
Breaks a problem in smaller identical problems.
An alternative to iteration (an iterative solution involves loops).

AN EXAMPLE OF RECURSION: SEQUENTIAL SEARCH VS BINARY SEARCH

Sequential Search: Starts at the beginning of the collection.
Checks every item in collection in order until target item is found.

Binary Search: Repeatedly halves collection and find the half with target item.
Uses a divide and conquer strategy.

RECURSIVE SOLUTIONS 2

DESCRIPTION OF A RECURSIVE SOLUTION

A recursive method calls itself.
Each recursive call solves an identical, but smaller, problem (**recurrence relation**).
Test for the **base case** (known case in a recursive definition) to stop recursive calls.
Eventually, one of the smaller problems must be the base case.

HOW TO CONSTRUCT RECURSIVE SOLUTIONS

Perform the following steps to construct a recursive solution:

1. define the problem in terms of a smaller problem of the same type;
2. ensure that each recursive call diminish the size of the problem;
3. define the instance of the problem that can serve as the base case;
4. ensure that as the problem size diminishes, you will reach the base case.

RECURSIVE SOLUTIONS 3

A RECURSIVE VALUED METHOD: THE FACTORIAL OF AN INTEGER A

Consider the problem of computing the factorial of an integer n.

$$factorial(n) = \begin{cases} 1, & n = 0 \\ n \times (n-1) \times (n-2) \times \cdots \times 1, & n > 0 \end{cases}$$

To define **factorial(n)** recursively you need to define it in terms of the factorial of a smaller number.

$$factorial(n) = n \times factorial(n-1), \qquad n > 0$$

This is an example of **recurrence relation**.
This recursive definition of **factorial(n)** still lacks one key element: the **base case**.

RECURSIVE SOLUTIONS 4

A RECURSIVE VALUED METHOD: THE FACTORIAL OF AN INTEGER B

The base case for the factorial method is **factorial(0)** which is equal to 1.

Because **n** originally is greater than or equal to **0**, and each call to **factorial** decrements **n** by **1**, we will always reach the **base case**.

So, adding the **base case** to the **recurrence relation**, the **complete recursive definition** of the factorial method is:

$$factorial(n) = \begin{cases} 1, & n = 0 \\ n \times factorial(n-1), & n > 0 \end{cases}$$

RECURSIVE SOLUTIONS 5

A RECURSIVE VALUED METHOD: THE FACTORIAL OF AN INTEGER C

Comparison between iterative and recursive implementations of the factorial of n.

```
public class FactorialExample {

   public static int factorialIterative( int n ) {
      int tempN = n; int result = 1;
      while( tempN > 1 ) { result *= tempN; tempN--; }
      return result; }

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

}
```

RECURSIVE SOLUTIONS 6

THE BOX TRACE

The box trace is a systematic way to **trace the actions of a recursive method**, where each box roughly corresponds to an activation record.

An **activation record** (i.e. a box) contains the **local environment of the recursive method at the time of and as a result of the call to the method**.

ELEMENTS OF A METHOD LOCAL ENVIRONMENT

The local environment of a method includes:

1. the **local variables** of the method,
2. a copy of the **actual value arguments**,
3. a **return address** in the calling routine,
4. the **value of the method** itself.

```
n = 3
A: fact( n-1 ) = ?
return ?
```

RECURSIVE SOLUTIONS 7

HOW TO BUILD A BOX TRACE A

Label each recursive call in the recursive method:

1. Distinguish each recursive call in the recursive method with a different label. These labels keep track of the return point to which each call musts return. The example below shows a recursive method with a recursive call labeled:

```
if( n == 0 ) {
   return 1; }
else {
   return ( n * fact( n - 1 ) ); } } // Label A.
```

RECURSIVE SOLUTIONS 8

HOW TO BUILD A BOX TRACE B

Each time a method is called, a new box represents its local environment:

2. Represent each recursive call during execution by a box showing the **method's local environment**. Each box will contain:
   a. the values of the references and primitive types of the input arguments;
   b. the local variables of the recursive method;
   c. a labeled (**step 1**) placeholder for the return value of each recursive call;
   d. the value of the method itself.

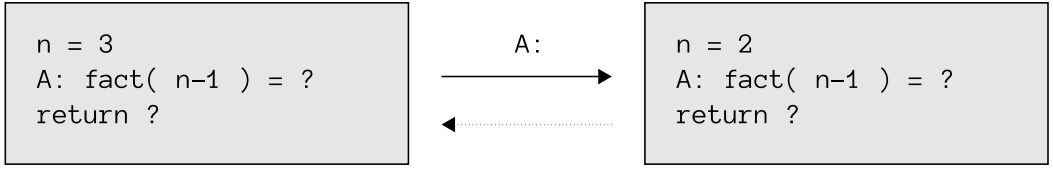
Note: When we create a box, we only know the values of the input arguments. We fill the values of other items as we determine them executing the method.

```
n = 3
A: fact( n-1 ) = ?
return ?
```

HOW TO BUILD A BOX TRACE C

Connect the boxes accordingly to the recursive calls made:

3. Draw an **arrow** from the statement that initiates the recursive process to the first box. Then, when you create a new box after a recursive call (see **step 2**) you draw an arrow from the box that makes the call to the newly created box. **Label each arrow** to correspond to the label of the recursive call (see **step 1**); this label indicates exactly where to return after the call completes.

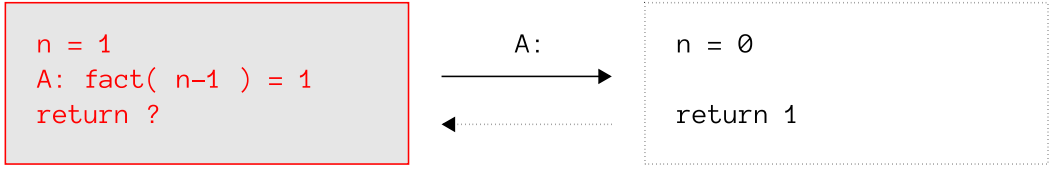


HOW TO BUILD A BOX TRACE D

Execute the methods and find the final result of the recursive calls:

4. After step 1, 2 and 3, you can execute the method. Each reference to an item in its local environment references the corresponding value in current box.

5. On exiting the method, cross off the current box (**dashed border**) and follow its arrow back (**dashed line**) to the box that called the method. This box now becomes the current box (**red color**), and the label on the arrow specifies the location to continue execution. **Remember to substitute the returned value!**



RECURSIVE SOLUTIONS 11

A RECURSIVE VOID METHOD: WRITING A STRING BACKWARD A

Consider the problem of writing a string backward. We should start defining our solution for a string of length **n**, in terms of a solution for a smaller string:

$$reverse(\text{"apple"}) = reverse(\text{"pple"}) + \text{"a"} = \cdots = \text{"elppa"}$$

or

$$reverse(\text{"apple"}) = \text{"e"} + reverse(\text{"appl"}) = \cdots = \text{"elppa"}$$

Note: The left and right calls of **reverse** have input arguments with different lengths!

One of these definitions is our **recurrence relation**.

A RECURSIVE VOID METHOD: WRITING A STRING BACKWARD B

Then we should find our **base case** (or cases), with a known solution.

Note: This is required to allow the recursion to stop!

Example: Since it is easy to reverse an **empty string**, or a **single character**, we can use these as our base cases (in both cases the solution is the same input string).

RECURSIVE SOLUTIONS 13

A RECURSIVE VOID METHOD: WRITING A STRING BACKWARD C

Comparison between **iterative** and **recursive** reversing of a string.

```
public class WriteBackwardExample {

   public static void writeBackwardIterative( String s ) {
      for( int i = s.length() - 1; i >= 0; i-- ) { System.out.print( s.charAt(i) ); } }

   public static void writeBackward1Recursive( String s ) {
      if( s.length() > 0 ) {
         System.out.print( s.substring( s.length() - 1 ) );
         writeBackward1Recursive( s.substring( 0, s.length() - 1 ) ); } }

   public static void writeBackward2Recursive( String s ) {
      if( s.length() > 0 ) {
         writeBackward2Recursive( s.substring( 1, s.length() ) );
         System.out.print( s.substring( 0, 1 ) ); } }

}
```

COUNTING THINGS 1

INTRODUCTION OF ANOTHER TYPE OF PROBLEMS: COUNTING THINGS

These problems require you to **count** certain events or things, usually contain **more than one base case**, and are good **examples of inefficient recursive solutions**.

MULTIPLYING RABBITS: THE FIBONACCI SEQUENCE A

Count pairs of rabbits of a farmer at a specific month, assuming the following facts:

1. rabbits never die;
2. a rabbit reaches sexual maturity at the beginning of the 3rd month;
3. rabbits are always born in male-female pairs;
4. every month, each sexually mature pair gives birth to exactly 1 pair;
5. suppose the farmer starts with 1 male-female pair at month 1.

Question: How many pairs would there be in month 6, including births at month 6?

Answer: M1(1), M2(1), M3(2=M2+M1), M4(3=M3+M2), M5(5=M4+M3), M6(8=M5+M4)

COUNTING THINGS 2

MULTIPLYING RABBITS: THE FIBONACCI SEQUENCE B

The **recurrence relation** for **rabbit(n)**, in terms of **rabbit(n-1)** and **rabbit(n-2)**, is:

$$rabbit(n) = rabbit(n-1) + rabbit(n-2), \qquad n \geq 3$$

Note: The solution of this problem is obtained solving **>1** smaller problem.

Choose base cases carefully: ensure all recursive calls will reach one base case.

Since our recurrence relation relies on **2 smaller problems with different size reductions**, we have the following **2 base cases**:

$$rabbit(1) = rabbit(0) = 1$$

COUNTING THINGS 3

MULTIPLYING RABBITS: THE FIBONACCI SEQUENCE C

Finally our complete **recursive definition** for counting the rabbits at month **n** is:

$$rabbit(n) = \begin{cases} 1, & 2 \geq n \geq 1 \\ rabbit(n-1) + rabbit(n-2), & n \geq 3 \end{cases}$$

Note: This series of numbers is known as the **Fibonacci sequence**, which models many naturally occurring phenomena.

COUNTING THINGS 4

MULTIPLYING RABBITS: THE FIBONACCI SEQUENCE D

Comparison between **iterative** and **recursive** methods to compute the **n^{th} Fibonacci term**.

```
public class FibonacciSequenceExample {

   public static int rabbitIterative( int n ) {
      if( n <= 2 ) { return 1; }
      else {
         int a = 1; int b = 1; int tmpA;
         for( int i = 3; i <= n; i++ ) { tmpA = a; a = b; b += tmpA; }
         return b; } }

   public static int rabbitRecursive( int n ) {
      if( n <= 2 ) { return 1; }
      else { return ( rabbitRecursive( n - 1 ) + rabbitRecursive( n - 2 ) ); } }

}
```

COUNTING THINGS 5

ORGANIZING A PARADE A

The rules to organize a parade are:

1. the parade will consist of **bands** and **floats** in a **single line**;
2. one band cannot be placed immediately after another (i.e. **no band-band**).

Question: If you have **n bands** and **n floats**, in how many ways can you organize a parade of length **n**?

Let:

$$P(n) = \text{the number of ways to organize a parade of length n}$$
$$F(n) = \text{the number of parades of length n that end with a float}$$
$$B(n) = \text{the number of parades of length n that end with a band}$$

Then: $$P(n) = F(n) + B(n)$$

COUNTING THINGS 6

ORGANIZING A PARADE B

Consider **F(n)**, the number of acceptable parades of length **n** ending with a **float**. A parade like these is any acceptable parade of length **n-1** plus a **float** at the end. So:

$$F(n) = P(n-1)$$

Consider **B(n)**, the number of acceptable parades of length **n** ending with a **band**. The only way a parade can end with a **band** is if the unit before the end is a **float**. So:

$$B(n) = F(n-1) = P(n-2)$$

So, the number of acceptable parades of length **n** is (**recurrence relation**):

$$P(n) = F(n) + B(n) = P(n-1) + F(n-1) = P(n-1) + P(n-2)$$

ORGANIZING A PARADE C

Considering the recurrence relation found, we have **2 base cases**:

$P(1) = 2 =$ the parades of size 1 are: "float", "band"
$P(2) = 3 =$ the parades of size 2 are: "float–float", "band–float", "float–band"

So our complete **recursive definition** is:

$$P(n) = \begin{cases} P(1) = 2 \\ P(2) = 3 \\ P(n) = P(n-1) + P(n-2), \qquad n \geq 3 \end{cases}$$

COUNTING THINGS 8

MR. SPOCK'S DILEMMA: CHOOSING K OUT OF N THINGS A

Question: How many different choices are possible for exploring **k** out of **n** planets?

Let:

$$c(n,k) = \text{the number of groups of } k \text{ planets chosen from } n$$

However, if we analyze the problem in terms of a specific **Planet X**:

$$c(n,k) = (\text{the number of groups of } k \text{ planets that include } Planet\ X) + (\text{the number of groups of } k \text{ planets that do not include } Planet\ X)$$

MR. SPOCK'S DILEMMA: CHOOSING K OUT OF N THINGS B

Therefore, the number of ways to choose **k** out of **n** planets is the **sum** of:

- the number of ways to choose **k-1** out of **n-1** planets (when **Planet X** is among the **k** planets), **plus**
- the number of ways to choose **k** out of **n-1** things (when **Planet X** is among the other **n-k** planets).

So the **recurrence relation** is: $c(n,k) = c(n-1,k-1) + c(n-1,k)$

Whereas the base cases are the following:

$$c(k,k) = 1 = \text{the number of groups of } k \text{ planets chosen from } k, \quad \text{if } k = n$$
$$c(n,0) = 1 = \text{the number of groups of } 0 \text{ planets chosen from } n, \quad \text{if } k = 0$$
$$c(n,k) = 0 = \text{the number of groups of } k \text{ planets chosen from } n, \quad \text{if } k > n$$

MR. SPOCK'S DILEMMA: CHOOSING K OUT OF N THINGS C

Therefore, the complete **recursive definition** is:

$$c(n,k) = \begin{cases} 1 & k = 0 \\ 1 & k = n \\ 0 & k > n \\ c(n-1,k-1) + c(n-1,k) & 0 < k < n \end{cases}$$

```
public static int chooseKOutOfNThings( int k, int n ) {
   // Base cases.
   if( ( k == 0 ) || ( k == n ) ) { return 1; }
   else if( k > n ) { return 0; }
   // Recursive calls.
   else {
      return ( chooseKOutOfNThings( k-1, n-1 ) + chooseKOutOfNThings( k, n-1 ) ); } }
```

SEARCHING AN ARRAY 1

BINARY SEARCH A

Consider the problem of looking up a word in a dictionary.

- We can use a **sequential search**, looking at every word in the dictionary in order until we find our target word.

- Otherwise we can use a faster way to perform the search, using a **binary search**. In this case, we will repeatedly halve the dictionary and determine at each split the half containing the target word. Obviously to do this we rely on the fact that the dictionary is ordered.

SEARCHING AN ARRAY 2

BINARY SEARCH B

A **recursive solution for a binary search** is the following:

1. check the **base case:** an array with only 1 item:
    a. if so, if the single item is equal to target value return the item index;
2. otherwise:
    a. find the **middle index of the array**;
    b. determine **which half of the array** may contain the target value;
    c. apply the **binary search** to the proper half of the array.

The following are the **issues we need to consider** to implement a binary search:

1. How to **pass "half of the array" to the recursive calls** in the binary search?
2. How to **determine which half** of the array contains the target value?
3. What are the **base cases**?
4. How to **indicate the final result** of the binary search?

SEARCHING AN ARRAY 3

BINARY SEARCH C

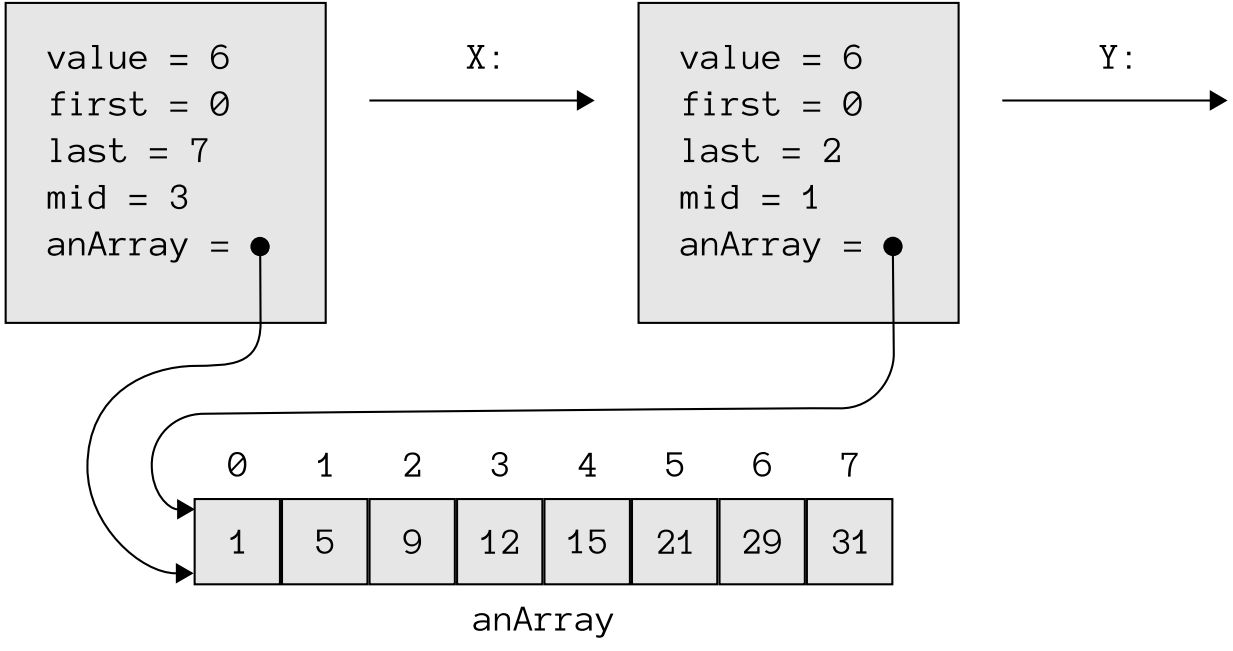
This is a method implementing a binary search (**recursive calls** are labeled **X** and **Y**):

```
public static int binarySearch( int anArray[], int first, int last, int value ) {
   int index;
   // Check if search range is valid.
   if( first > last ) { index = -1; }
   else {
      // Invariant: if value is in anArray, anArray[first] <= value <= anArray[last].
      int mid = ( first + last ) / 2;
      // Check if value is at anArray[mid], or determine the half containing value.
      if( value == anArray[mid] ) { index = mid; }
      else if( value < anArray[mid] ) {
         index = binarySearch( anArray, first, mid-1, value ); } // Search half 1: X.
      else {
         index = binarySearch( anArray, mid+1, last, value ); } } // Search half 2: Y.
   return index; // Return index where value is stored, or -1 if value was not found.
}
```

SEARCHING AN ARRAY 4

BINARY SEARCH D

Remember that **in Java an array is an object**, so each call to the binary search method uses a copy of a reference to the array (i.e. the array content is not copied).



SEARCHING AN ARRAY 5

FINDING THE KTH SMALLEST ITEM IN AN ARRAY A

Find the **kth** smallest item in an unordered array, without completely sorting the array.

Note: In this case the **reduction in size** of the problem to be solved will be **unpredictable**, because it depends on the items in the array. This is due to the fact that the "global" rank of an item cannot be derived by its "local" rank in a specific partition of the array.

The **recursive solution** proceeds as follows:

1. select a **pivot item** in the array;
2. cleverly **partition** the array accordingly to this **pivot item**, for example:
    a. partition $\mathbf{S_1}$ containing all the items **< pivot item**, and
    b. partition $\mathbf{S_2}$ including all the items **≥ pivot item**;
3. recursively **apply the strategly to one of the partitions**.

SEARCHING AN ARRAY 6

FINDING THE KTH SMALLEST ITEM IN AN ARRAY B

Suppose that we are able to find a **pivot item** to create partitions $\mathbf{S_1}$ and $\mathbf{S_2}$.
This induces 3 smaller problems, such that solving one of these will also solve the original problem:

1. if $\mathbf{S_1}$ contains **k** or more items, $\mathbf{S_1}$ contains the **k** smallest items of the array and the $\mathbf{k^{th}}$ smallest item is in $\mathbf{S_1}$; this case occurs if: **k < (pivotIndex – first + 1)**

2. if $\mathbf{S_1}$ contains **k-1** items, the $\mathbf{k^{th}}$ smallest item must be the pivot item **p**; this is the **base case**, and it occurs if: **k = (pivotIndex – first + 1)**

3. if $\mathbf{S_1}$ contains fewer than **k-1** items, the $\mathbf{k^{th}}$ smallest item in the array is in $\mathbf{S_2}$; this case occurs if: **k > (pivotIndex – first + 1)**

SEARCHING AN ARRAY 7

FINDING THE KTH SMALLEST ITEM IN AN ARRAY C

The complete **recursive definition** is the following.

Let: $kMin(k, a, f, l) = \text{the } k^{th} \text{ smallest item in array } a\ [f \ldots l]$

Then:

$$kMin(k, a, f, l) = \begin{cases} kMin(k, a, f, p-1) & k < (p - f + 1) \\ p & k = (p - f + 1) \\ kMin(k - (p - f + 1), a, p + 1, l) & k > (p - 1 + 1) \end{cases}$$

where **p** is the **index** of the **pivot item**.

SEARCHING AN ARRAY 8

FINDING THE KTH SMALLEST ITEM IN AN ARRAY D

The only questions that remain are:

Question: How to choose the pivot item?

Answer: The choice of the pivot item is arbitrary. Any pivot item in the array will work. However, the sequence of choices will affect the number of recursive calls to reach the base case.

Question: How to partition the array about the chosen pivot item?

Answer: Different strategies will be discussed in CS-203.

ORGANIZING DATA 1

THE TOWERS OF HANOI A

The "Towers of Hanoi" puzzle consists of **n** disks and **3** poles:

- **A** (i.e. the **source**),
- **B** (i.e. the **destination**), and
- **C** (i.e. the **spare**).

The **disks are of different sizes** and have holes in the middle so they can fit on the poles. Because of their weight, the **disks can be placed only on top of disks larger than themselves**.

Initially all the disks are placed on pole **A**.

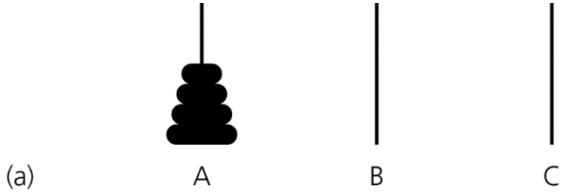
The puzzle consists in **moving the disks, one by one, from pole A to pole B exploiting also pole C**.

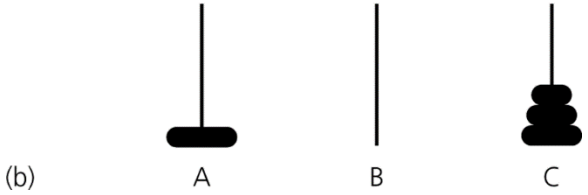
ORGANIZING DATA 2

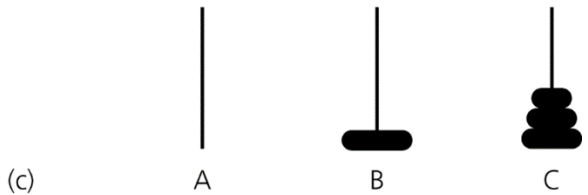
THE TOWERS OF HANOI B

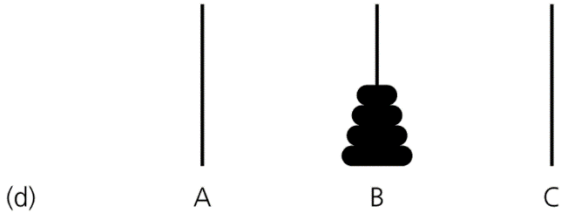
The **recursive solution** for the "Towers of Hanoi" puzzle:

a. the initial state;
b. move **n-1** disks from pole **A** to pole **C**;
c. move **1** disk from pole **A** to pole **B**;
d. move **n-1** disks from pole **C** to pole **B**.









THE TOWERS OF HANOI C

The **step-by-step recursive solution** of the "Towers of Hanoi" puzzle is the following:

1. if there is only **1** disk (**n=1**), move it from pole **A** to pole **B** (**base case**);

2. if there is more than **1** disk (**n>1**):
   a. ignore bottom disk and solve for **n-1** disks, using pole **A** as the **source** and pole **C** as the **destination** with pole **B** as the **spare** (see figure **b**);
   b. then, **n-1** disks are on pole **C** and the largest disk is on pole **A**, so you can solve for **n=1** moving the largest disk from pole **A** to pole **B** (see figure **c**);
   c. finally, move the **n-1** disks from pole **C** to pole **B**. That is, solve the problem using: pole **C** as the **source**, pole **B** as the **destination**, and pole **A** as the **spare** (see figure **d**).

ORGANIZING DATA 4

THE TOWERS OF HANOI D

Recursive implementation in Java to solve the "Towers of Hanoi" puzzle:

```
public static void solveTowersOfHanoi( int n, char source, char dest, char spare ) {
   // Base case.
   if( n == 1 ) {
      System.out.println( "Move top disk from " + source + " to " + dest ); }
   // Recursive calls.
   else {
      solveTowersOfHanoi( n-1, source, spare, dest ); // X.
      solveTowersOfHanoi( 1, source, dest, spare ); // Y.
      solveTowersOfHanoi( n-1, spare, dest, source ); } // Z.
}
```

RECURSION AND EFFICIENCY 1

Some recursive solutions we have seen are inefficient (exceptions are the **binarySearch** and **solveTowersofHanoi** functions).
Two factors contribute to the inefficiency of some recursive solutions:

1. overhead associated with method calls:
    a. this overhead affects method calls in general (in high-level languages);
    b. recursive methods quickly generate several recursive calls, so recursive methods increase a lot this overhead;
    c. input params passed to recursive methods increase the overhead too;

2. inherent inefficiency of some recursive algorithms:
    a. multiple computation of the same value (e.g. **rabbitRecursive** function).

Note: Recursion is truly valuable when a problem has no simple iterative solutions.

RECURSION AND EFFICIENCY 2

CONVERTING A RECURSIVE SOLUTION TO AN ITERATIVE SOLUTION

Sometimes, it may be easier to discover a recursive solution than an iterative solution. If, in these cases, the iterative solution is more efficient a conversion may be required.

The conversion is easy if there is only 1 recursive call in the recursive method.
The conversion may be difficult if there are multiple recursive calls.

The conversion is really simple if the recursive call is the last action of the method. This kind of recursion is called **tail recursion** (e.g. **writeBackward1Recursive**).

Usually in these cases (i.e. iterative functions converted from recursive functions) the iterative solution is more efficient than the recursive solution.

Note: Some compilers auto-convert tail recursive methods to iterative methods!