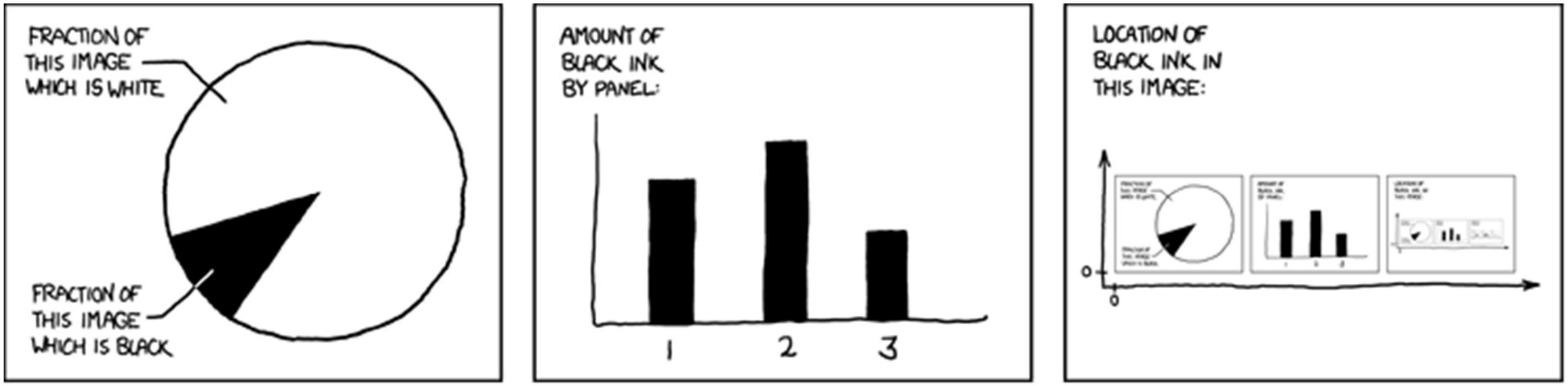
Introduction to Recursion

1



And Recursive Algorithms

Different Views of Recursion

- **Recursive Definition**: n! = n * (n-1)!
  (non-math examples are common too)
- **Recursive Procedure**: ________________
- **Recursive Data Structure**: a data structure that contains a pointer to an instance of itself:

```
public class ListNode {
    Object nodeItem;
    ListNode next, previous;
    …
}
```

Recursion in Algorithms

- Recursion is a technique that is useful
  - for defining relationships, and
  - for designing algorithms that implement those relationships
- Recursion is a natural way to express many algorithms
- For recursive data-structures, recursive algorithms are a natural choice

What Is Recursion?

- A definition is recursive if ______________________
  - We use them in grammar school e.g. what is a noun phrase?
    - a noun
    - an adjective followed by a noun phrase
  - Descendants
    - the person's children
    - all the children's descendants

Recursion…

- Questions to ask yourself
  - How can we reduce the problem to smaller version of the same problem?
  - How does each call make the problem smaller?
  - What is the ______________?
  - Will you always reach it?

Other Recursive Definitions in Mathematics

- Factorial:

  $n! = n\,(n-1)!$ and $0! = 1! = 1$

- Fibonacci numbers:

  $F(0) = F(1) = 1$

  $F(n) = F(n-1) + F(n-2)$ for $n > 1$

- Note base case
  - Definition can't be completely self-referential
  - Must eventually come down to something that's ____________ ____________

Question…

- I know the steps needed to write a simple recursive method in Java

1. Strongly Agree
2. Agree
3. Disagree
4. Strongly Disagree

Recursive Factorial


```
public static int factorial (int n) {
  if (n == 1)  //BASE CASE
          return 1;
  else
          return n * factorial(n-1);
}
```

- Exercise: trace execution (show method calls) for **n=5**

Recursive Factorial (for n=5)

return 5 * <u>factorial(4)</u>

  return 4 * <u>factorial(3)</u>

    return 3 * <u>factorial(2)</u>

      return 2 * <u>factorial(1)</u>

        return 1

So … going bottom to top:

return 2 * (1)

  return 3 * (2 * 1)

    return 4 * (3 * 2 * 1)

      return 5 * (4 * 3 * 2 * 1)

        END

Result: 5*4*3*2*1 = 5!

Why Do Recursive Methods Work?

- **Activation Records** on the **Run-time Stack** are the key:
  - Each time you call a function (any function) you get a new activation record
  - Each activation record contains a copy of all local variables and parameters for that invocation
  - The activation record remains on the stack until the function returns, then it is destroyed
- Try yourself: use your IDE's debugger and put a breakpoint in the recursive algorithm. Look at the call-stack

Example, n=4 (Run-time stack)

Broken Recursive Factorial


```
public static int Brokenfactorial(int n){
    int x = Brokenfactorial(n-1);
    if (n == 1)
        return 1;
    else
        return n * x;
}
```

- What's wrong here? Trace calls "by hand"
  - BrFact(2) -> BrFact(1) -> BrFact(0) -> BrFact(-1) -> BrFact(-2) -> ...
  - Problem: we do the recursive call **first** before checking for the base case
  - **Never stops!** Like an infinite loop!

Recursive Design

- Recursive methods/functions require:
  1. One or more (non-recursive) ______________ that will cause the recursion to end

     ```
     if (n == 1) return 1;
     ```

  2. One or more ______________________ that operate on smaller problems and get you *closer* to the base case

     ```
     return n * factorial(n-1);
     ```

- Note: The base case(s) should always be checked __________ the recursive call

How to Think/Design with Recursion

- Many people have a hard time writing recursive algorithms
- The key: focus only at the current "stage" of the recursion
  - Handle the base case, then…
  - Decide what recursive-calls need to be made
    - Assume they work (*as if by magic*)
  - Determine how to use these calls' **results**

Recursion Example: List Processing

- Is an item in a list? First, get a reference current to the first node
  - (Base case) If current is null, return false
  - (Base case #2) If the first item equals the target, return true
  - (Recursive case – might be 1n the remainder of the list)
    - current = current.next
    - return result of recursive call on *new* current

Recursion vs. Iteration

- **Interesting fact**: Any recursive algorithm can be re-written as an iterative algorithm (loops)
- Not all programming languages support recursion: e.g. COBOL, early FORTRAN
- Some programming languages rely on recursion **heavily**: e.g. LISP, Prolog, Scheme

To Recurse or Not To Recurse?

- Recursive solutions often seem elegant
- Sometimes recursion is an efficient design strategy
- But sometimes it's definitely not
  - Important! we can design a recursive solution and implement it non-recursively
  - Many recursive algorithms can be re-written non-recursively
    - Use an explicit stack
    - Remove tail-recursion (compilers often do this for you)

Recursive Fibonacci method

- *"This is elegant code, no?"* ☺

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

- Let's start to trace it for fib(5)

Trace of fib(5)

- For fib(5), we call fib(4) and fib(3)
  - For fib(4), we call fib(3) and fib(2)
    - For fib(3), we call fib(2) and fib(1)
      - For fib(2), we call fib(1) and fib(0). Base cases!
      - fib(1). *Base case!*
    - For fib(2), we call fib(1) and fib(0). Base cases!
  - For fib(3), we call fib(2) and fib(1)
    - For fib(2), we call fib(1) and fib(0). Base cases!
    - fib(1). *Base case!*

Fibonacci: recursion is a **bad choice**

- Note that subproblems (like fib(2)) repeat, and solved again and again
  -
  - For this problem, better to store partial solutions instead of recalculating values repeatedly
  - Turns out to have *exponential time-complexity!*

Non-recursive Fibonacci

- Two **bottom-up** iterative solutions:
  - Create an array of size n, and fill with values starting from 1 and going up to n
  - Or, have a loop from small values going up, but
    - only remember two previous Fibonacci values
    - use them to compute the next one
    - (See next slide)

Iterative Fibonacci


```
long fib(int n) {
  if ( n < 2 ) return 1;
    long answer;
    long prevFib=1, prev2Fib=1; // fib(0) & fib(1)
    for (int k = 2; k <= n; ++k) {
            answer = prevFib + prev2Fib;
            prev2Fib = prevFib;
            prevFib = answer;
  }
  return answer;
}
```

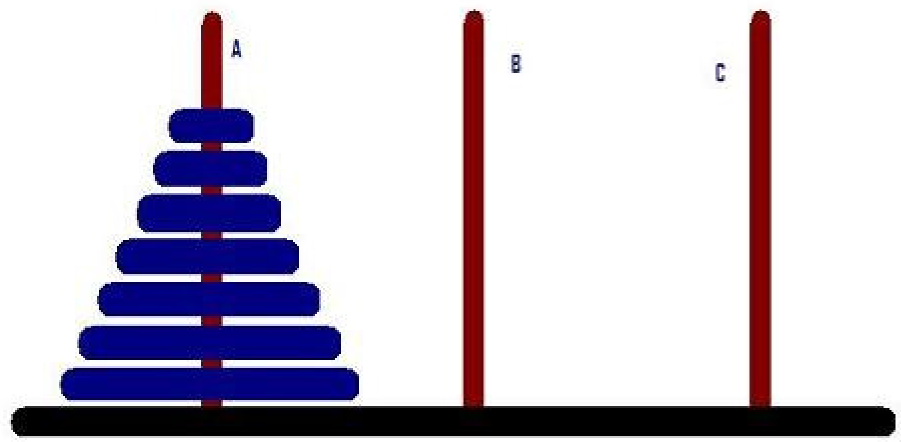
Iterative Factorial


```
int fact(int num) {
       int tmp = 1;
       for (int i=1; i<=num; i++) {
              tmp *= i;
       }
       return tmp;
}
```

Exercise: Trace fact(5)

Other Recursive Examples

- Towers of Hanoi
- Euclid's Algorithm
- General activities like
  - Is string a Palindrome?
  - Reverse a String
  - …



Towers of Hanoi

- The objective is to transfer entire tower A to the peg B, moving only one disk at a time and never moving a larger one onto a smaller one
- The algorithm to transfer n disks from A to B in general: We first transfer ***n - 1*** smallest disks to peg C, then move the largest one to the peg B and finally transfer the ***n - 1*** smallest back onto largest (peg B)
- The number of necessary moves to transfer ***n*** disks can be found by $T(n) = 2^n - 1$

Euclid's Algorithm

- Calculating the **greatest common divisor** (gcd) of two positive integers is the largest integer that divides evenly into both of them
- E.g. greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68
- Logic: If $p > q$, the gcd of p and q is the same as the gcd of q and $p \% q$, where % Remainder operator

Euclid's Algorithm

// recursive implementation

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

// non-recursive implementation

```
int gcd2(int p, int q) {
  while (q != 0) {
    int temp = q;
    q = p % q;
    p = temp;
  }
  return p;
}
```

Palindrome


```
public static boolean isPal(String s)    {
        if(s.length() == 0 || s.length() == 1)
            return true; //if length =0 OR 1 then it is
        if(s.charAt(0) == s.charAt(s.length()-1)) // see note
            return isPal(s.substring(1, s.length()-1));
             //if its not the case than string is not
        return false;
}
```

Note: check for first and last char of String, if they are same then do the same thing for a substring with first and last char removed. Carry on until your string completes or condition fails