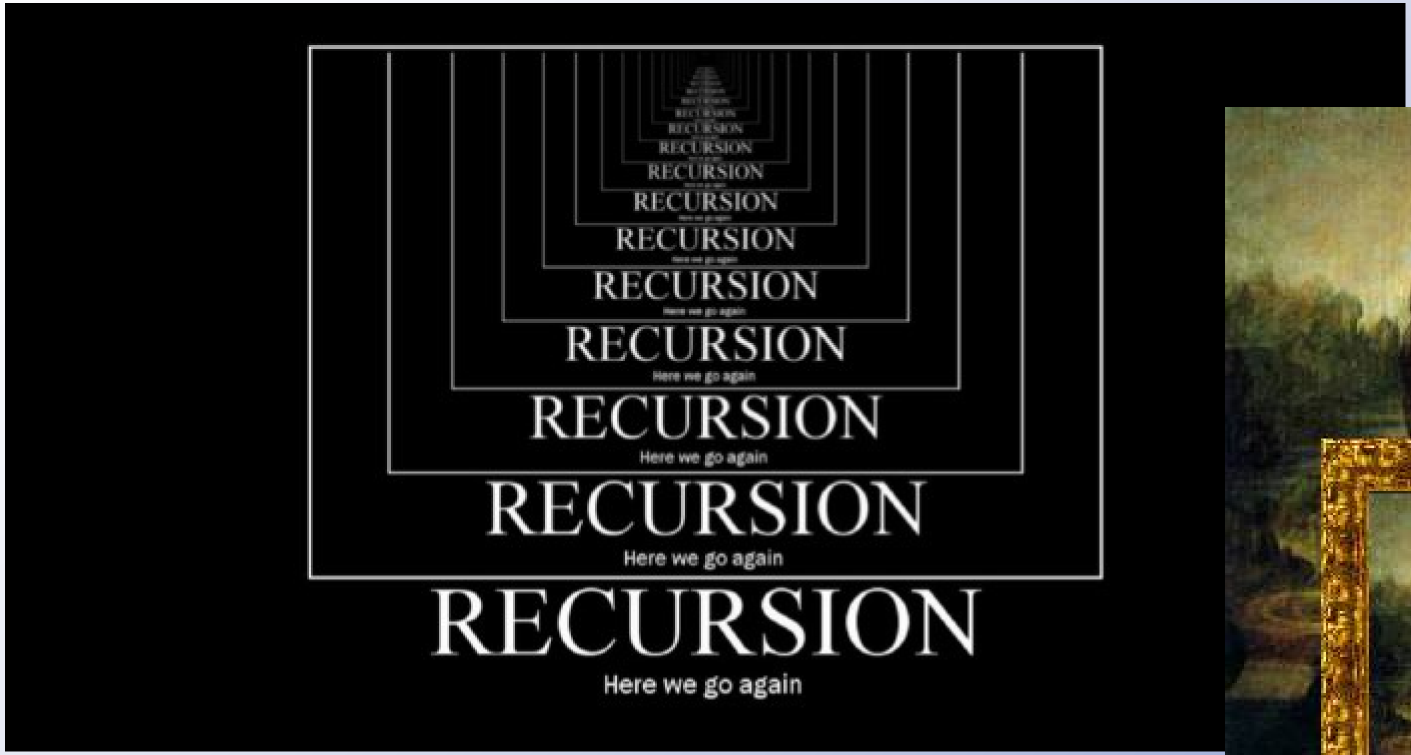
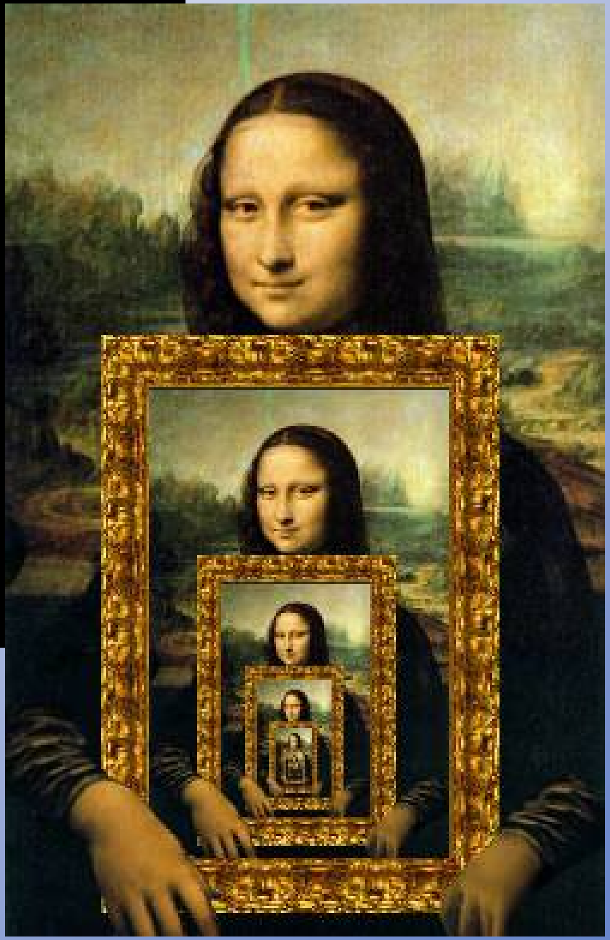
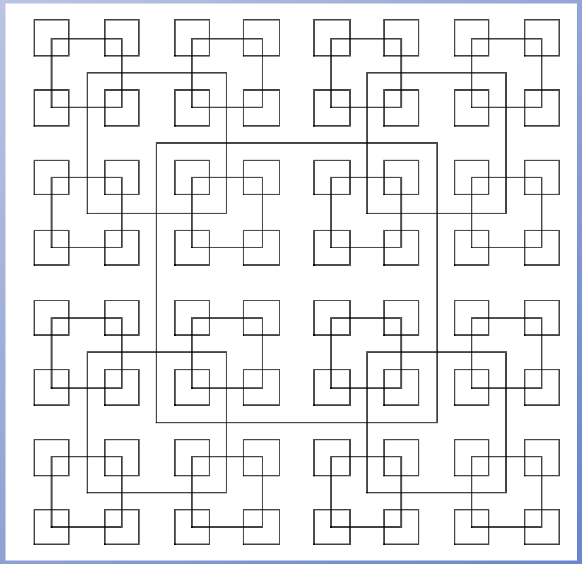
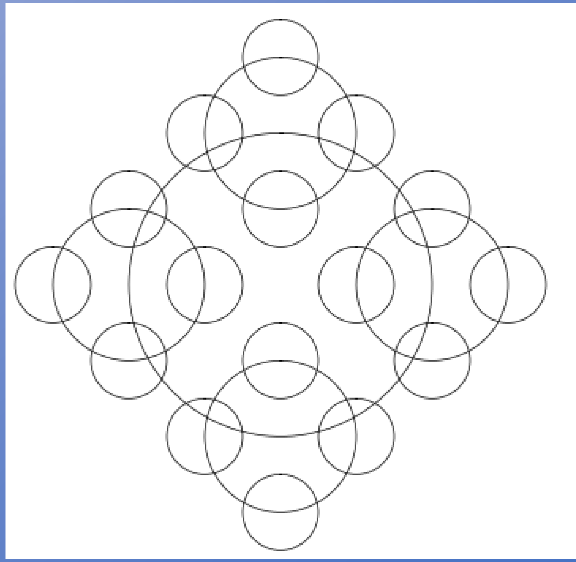
Recursion

Recursion


RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again







Recursion

- **Recursion** is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
- Recursion solves such problems by using functions that call themselves from within their own code.
- The approach can be applied to many types of problems,
- Recursion is one of the central ideas of computer science.

Is recursion ever necessary?

The short answer is:

NO

Anything that can be achieved with recursion can be achieved with iteration

Iteration is usually more efficient than recursion

When *should* we use recursion (non-exhaustive list):

1. When the algorithm to implement is defined recursively
   - Mergesort
   - Quicksort
   - Binary search
2. To traverse a data structure that is defined recursively
   - Linked lists
   - Binary search trees
3. To explore all possibilities
   - Permutations – find all the permutations of the characters in a string
     - p('cat') = 'cat', 'cta', 'act', 'atc', 'tca', 'tac'
   - Combinations - find all the binary numbers that have n digits
     - b(3) = '000','001','010','011','100','101',110','111'
4. To search incrementally and exhaustively for solutions to a problem (a.k.a backtracking):
   - Subset sum: is there a subset of a set of integers S that adds up to g?
     - Subsetsum({2,5,8,12}, 4) = No
     - Subsetsum({2,5,8,12},17) = Yes (subset = {5,12})

Elements of recursion:

1. Bases case(s) – You must ALWAYS have some base cases which can be solved without recursion
2. Making progress-For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress towards the base case
3. Design rule – Assume all recursive calls work, then show that the original call will work – **Ignore the fact that the function is recursive!**
4. Compound interest rule – Never duplicate work by solving the same instance of a problem in separate recursive calls

More on the design rule

What do we mean when we say: 'assume all recursive calls work'?

1. They **return** the expected value(s), according to the definition of the problem

or

2. They produce the expected **output**, according to the definition of the problem

Remember:

Every call to a recursive function has its own local variables (even if the have the same name in all calls)

Recursive calls cannot change the values of local variables in the calling function - this is the most common error when writing recursive functions!

What does the **return** statement do?

Misunderstanding of **return** statements is another source of problems for beginning programmers

Do you understand exactly what **return** does?

A **return statement** **ends** the execution of a function and **returns** control to the calling function. Execution resumes in the calling function at the point immediately following the call. A **return statement** may also **return** a value to the calling function.

Elements of recursion:

1. Bases case(s) – You must ALWAYS have some base cases which can be solved without recursion
2. Making progress-For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress towards the base case
3. Design rule – If all recursive calls work, then the original call will work (assume recursive calls work, show original call will work) – **Ignore the fact that the function is recursive!**
4. Compound interest rule – Never duplicate work by solving the same instance of a problem in separate recursive calls

Checking if all elements of recursion are satisfied

An example: factorial

```
1 def factorial(n):
2     if n<=0:
3         return 1          # (1) Base case – solved without recursion
4     else:
5         return n*factorial(n-1)   # (2) Making progress - recursive call is made to a simpler instance of the problem
```

(3) Design rule:
Assume factorial(n-1) returns the factorial of n-1 (DON'T TRACE!)
Then the function works correctly, since $n! = n(n-1)!$ (line 5)

(4) Compound interest rule:
There's no repeated computation

Checking if all elements of recursion are satisfied

An example: adding the digits of an integer

```
def sum_digits(n):
    if n<=0:
        return 0    # (1) Base case – solved without recursion
    else:
        return n%10 + sum_digits(n//10)    # (2) Making progress - recursive call is made to a simpler instance of the problem
```

(3) Design rule:
Assume sum_digits(n//10) returns the sum of all the digits except for the last one, then the function works correctly, since n%10 is the last digit

(4) Compound interest rule:
There’s no repeated computation

Checking if all elements of recursion are satisfied

An example: Fibonacci numbers

```
def fib(n):
    if n<2:            # (1) Base case – solved without recursion
        return n
    else:
        return fib(n-1) + fib(n-2)   # (2) Making progress - recursive calls are
                                     # made to simpler instances of the problem
```

(3) Design rule:
Assume fib(n- 1) and fib(n-2) return the correct values, then the function works correctly, since fib(n) = fib(n-1) + fib(n-2)

(4) Compound interest rule:
It breaks the compound interest rule. For example, fib(n-2) will be called by both fib(n) and fib(n-1). Thus the function will run in exponential time; it will never finish for most values of n.

Checking if all elements of recursion are satisfied

An example: adding the elements of a list

```
def sum_list(L):
    s = 0
    if len(L)==0:           # (1) Base case – solved without recursion
        return s
    s += L[0]
    sum_list(L[1:])         # (2) Making progress - recursive call is
    return s                #     made to a simpler instance of the problem
```

(3) Design rule:

Assume `sum_list(L[1:])` returns the sum of L from index 1 to the end. Then the function does not work, since it returns s, which is equal to L[0].

(4) Compound interest rule:

There's no repeated computation

Checking if all elements of recursion are satisfied

An example: adding the elements of a list

```
def sum_list(L):
    s = 0
    if len(L)==0:         # (1) Base case – solved without recursion
        return s
    s += L[0]
    s += sum_list(L[1:])  # (2) Making progress - recursive call is
                          # made to a simpler instance of the problem
    return s
```

(3) Design rule:
Assume `sum_list(L[1:])` returns the sum of L from index 1 to the end. Then the function works correctly, since the sum of L is L[0] plus the sum of L[1:].

(4) Compound interest rule:
There’s no repeated computation