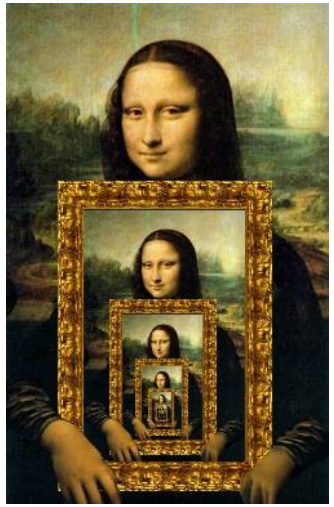
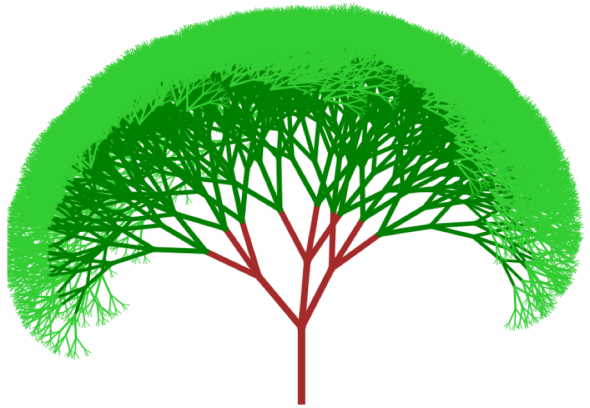
Recursion

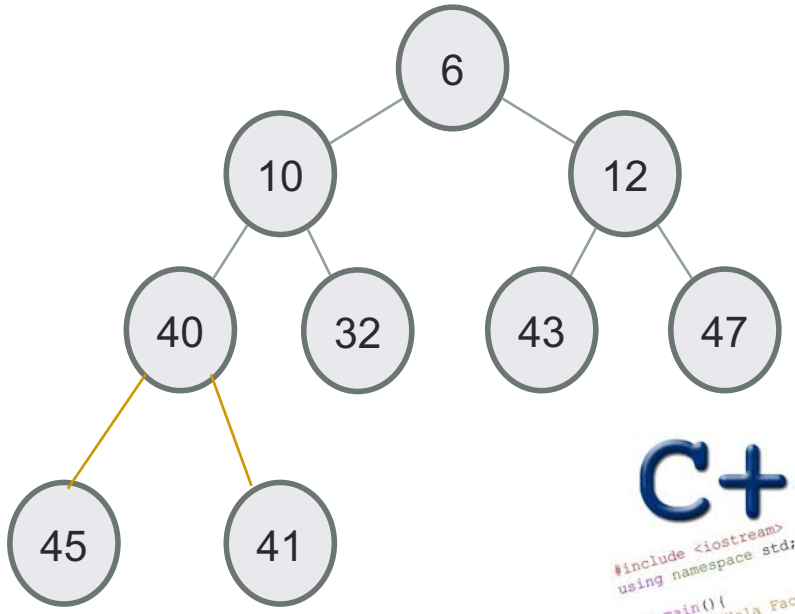
a.k.a., CS's version of mathematical induction



As close as CS gets to magic

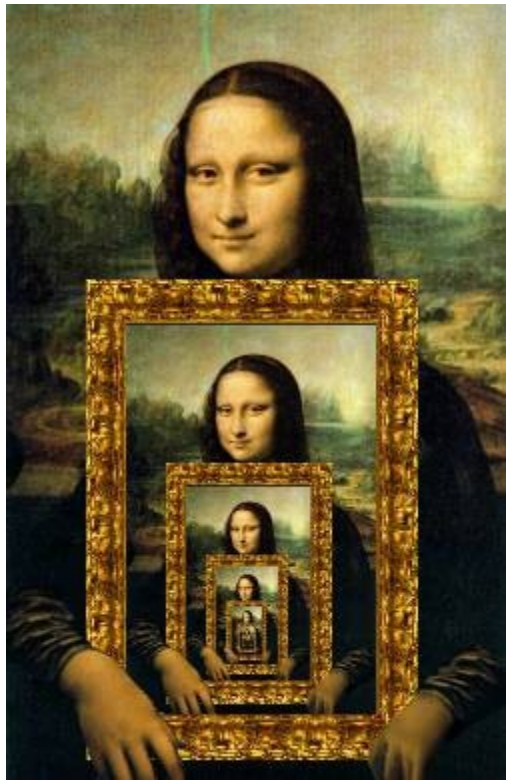
Problem Solving with Computers-I





Let recursion draw you in….

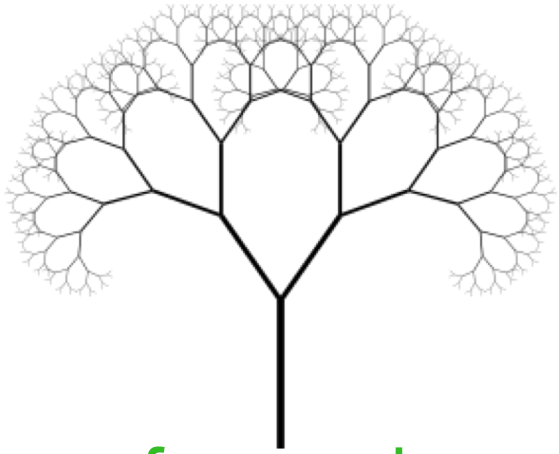
- Recursion occurs when something is described in terms of itself



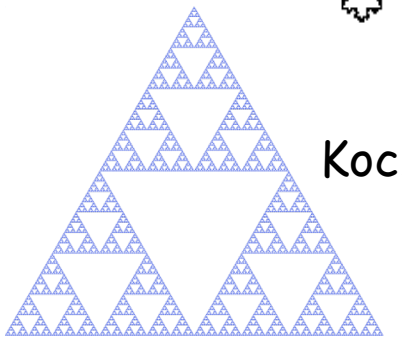
Recursive names

GNU IS NOT UNIX

Fractals



Koch's snowflake

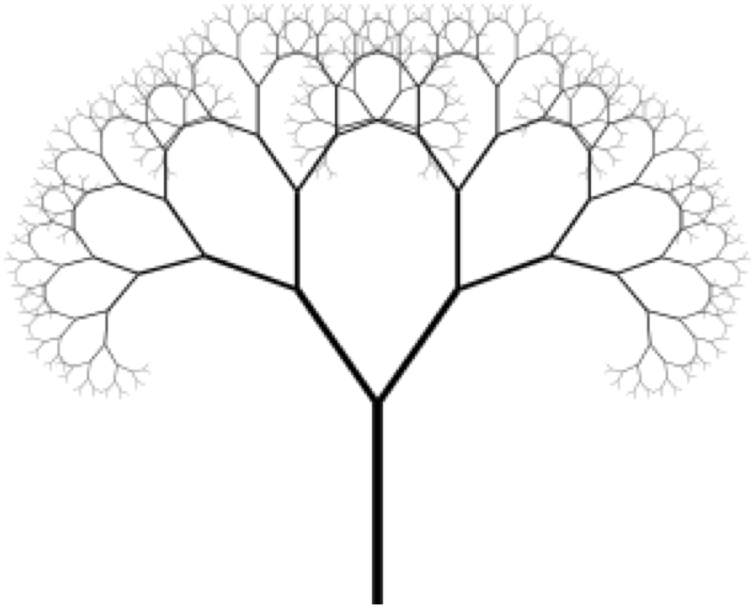


Sierpinski triangle

Visual representations of recursion

Recursion: A way of solving problems in CS

- Solve the simplest case of the problem
- Solve the general case by describing the problem in terms of a smaller version of itself



An everyday example:

To wash the dishes in the sink:

If there are no more dishes

you are done!

else:

Wash the dish on top of the stack

Wash the *remaining* dishes in the sink

Thinking *recursively*

```
N!  =  N * (N-1)! , if  N > 1
    =  1, if N <= 1
```

Recursion == ***self***-reference!

Designing Recursive Functions

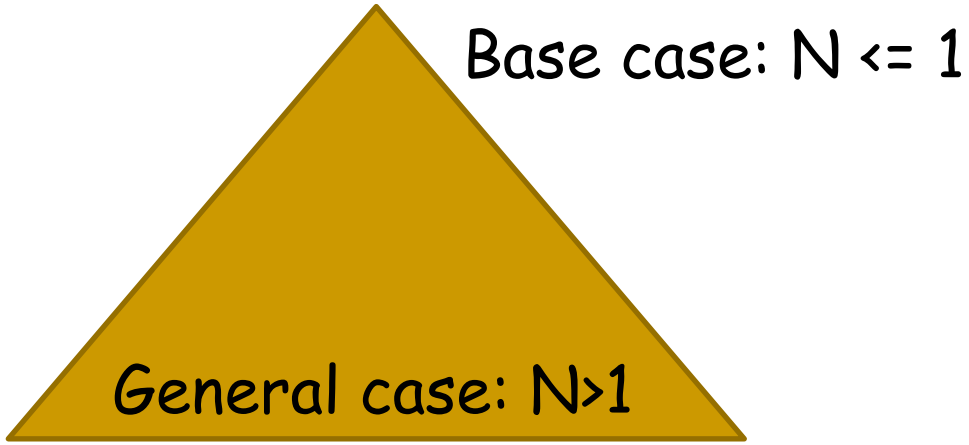
```
int fac(int N){
    if(N <= 1){
        return 1;
    }


}
```

Base case:

Solution to inputs where the answer is simple to solve

(top of the pyramid)



The pyramid of computation for recursive problems

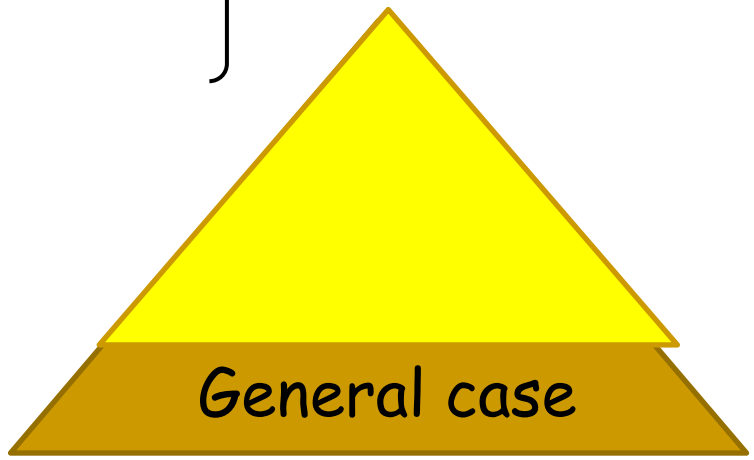
Designing Recursive Functions

```
int fac(int N){
    if(N <= 1){
        return 1;
    }else{
        double rest= fac(N-1);
        return N* rest;
    }

}
```

Base case

Recursive case



The pyramid of computation for recursive problems

Human: Base case and 1 step

Computer: Everything else

Warning: *this is legal!*

```
int fac(int N){
    return N* fac(N-1);
}
```

legal **!=** *recommended*

```
int fac(int N){
    return N* fac(N-1);
}
```

No ***base case*** -- the calls to **`fac`** will never stop!

Make sure you have a **base case**, *then* worry about the recursion...

Print the numbers 1 to N recursively

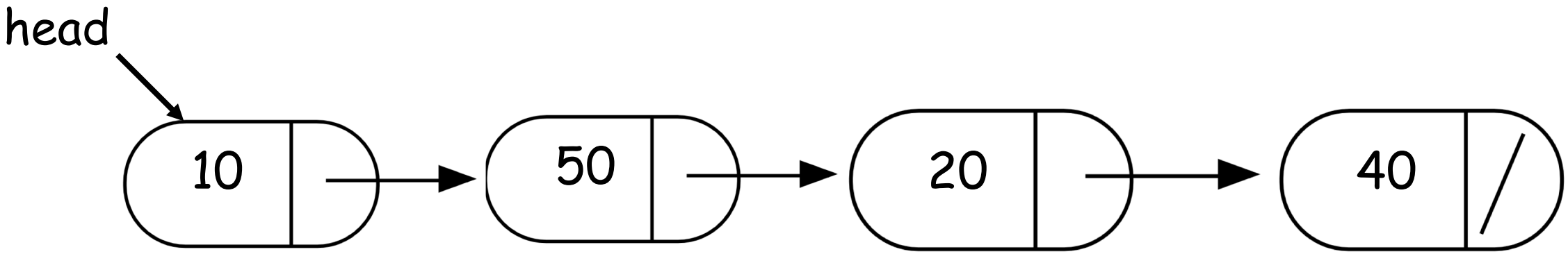
- Write a function to print the numbers from 1 to N (use recursion)

A new way of looking at inputs

Arrays:

- Non-recursive description: **a sequence of elements**
- Recursive description: **an element, followed by a smaller array**

Recursive description of a linked list



- Non-recursive description of the linked list: **chain of nodes**
- Recursive description of a linked-list: **a node, followed by a smaller linked list**

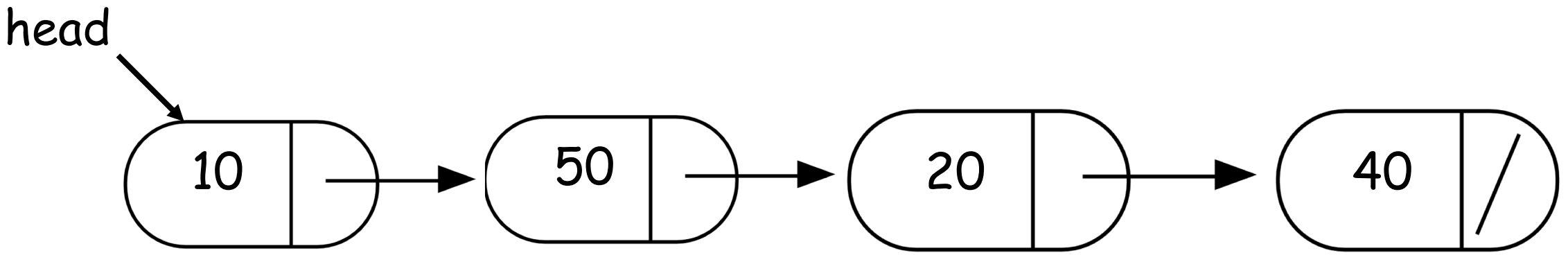
Designing recursive code: print all the elements of an array

Arrays:

- Recursive description: **an element, followed by a smaller array**

Designing recursive code: sum elements in a linked-list

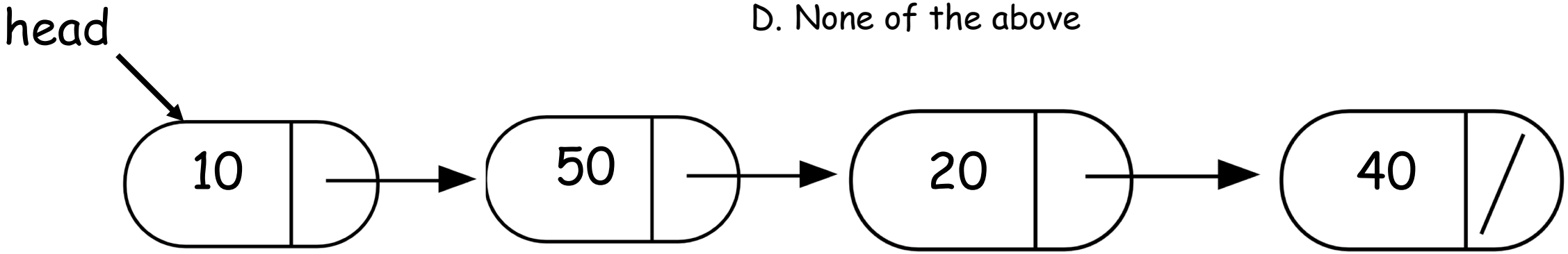
- Recursive description of a linked-list: **a node, followed by a smaller linked list**



What's in a base case?

What happens when we execute this code on the example linked list?

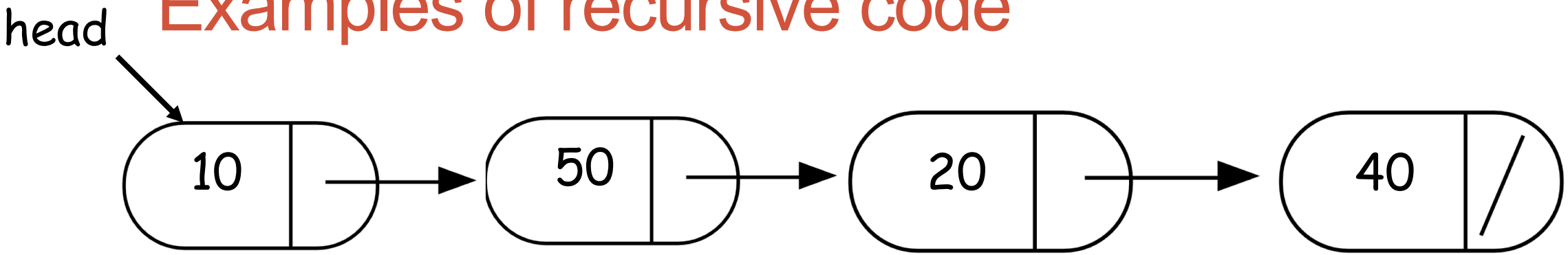
A. Returns the correct sum (120)
B. Program crashes with a segmentation fault
C. Program runs forever
D. None of the above




```
double sumList(Node* head){

    double sum = head->value + sumList(head->next);
    return sum;
}
```

Examples of recursive code




```
double sumList(Node* head){
    if(!head) return 0;
    double sum = head->value + sumList(head->next);
    return sum;
  }
```

Find the min element in a linked list

```
double min(Node* head){
    // Assume the linked list has at least one node
    assert(head);
    // Solve the smallest version of the problem









}
```

Helper functions

- Sometimes your functions takes an input that is not easy to recurse on
- In that case define a new function with appropriate parameters: This is your helper function
- Call the helper function to perform the recursion

For example

```
double sumLinkedLisr(LinkedList* list){
    return sumList(list->head); //sumList is the helper
    //function that performs the recursion.

}
```

Next time

- More practice with recursion
- Final practice