University of Southern California

Viterbi School of Engineering

Software Design

Recursion

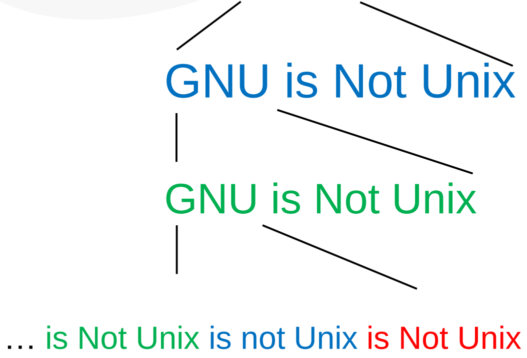
Reference: Professor Mark Redekopp’s Course Materials, Online Resources

Recursion

- **Recursion can be used as a fundamental programming technique to provide clean and elegant solutions to certain kinds of problems**
- **What is recursion: Defining an object, mathematical function, or computer function in terms of *itself***

GNU

- Makers of gedit, g++ compiler, etc.
- GNU = GNU is Not Unix



GNU is Not Unix

GNU is Not Unix

… is Not Unix is not Unix is Not Unix

- **Apparently recursive acronyms are humorous to programmers and hackers!**
  - **You are a hacker if you find recursions funny :D**

Example: LIST

```
A LIST is a:   number
       or a:   number  comma  LIST
```

- Question: What is the non-recursive form of

  LIST = `24, 88, 40, 37  ?`

non-recursive form:

```
number comma LIST
  24     ,    88, 40, 37

             number comma LIST
               88     ,    40, 37

                         number comma LIST
                           40     ,    37
```

GRAMMARS

Grammar Rules

- Languages have rules governing their syntax and meaning
- These rules are referred to as its grammar
- Programming languages also have grammars that code must meet to be compiled
  - Compilers use this grammar to check for syntax and other compile-time errors
  - Grammars often expressed as “productions/rules”
- ANSI C Grammar Reference:
  - http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#declaration

Simple Paragraph Grammar

Substitution	Rule
subject	"I" | "You" | "We"
verb	"run" | "walk" | "exercise" | "eat" | "play" | "sleep"
sentence	subject verb '.'
sentence_list	sentence | sentence_list sentence
paragraph	[TAB = \t] sentence_list [Newline = \n]

Example:

```
I run. You walk. We exercise.
subject verb. subject verb. subject verb.

sentence sentence sentence
sentence_list sentence sentence
sentence_list sentence
sentence_list
paragraph
```

Example:

```
I eat You sleep
Subject verb subject verb
Error
```

C++ Grammar

Rule	Expansion
expr	constant | variable_id | function_call | assign_statement | ‘(‘ expr ‘)’ | expr binary_op expr | unary_op expr
assign_statement	variable_id ‘=‘ expr
expr_statement	‘;’ | expr ‘;’

Example:

```
5 * (9 + max);
```

expr * (*expr* + *expr*) ;

expr * (*expr*) ;

expr * *expr*;

expr;

expr_statement

Example:

```
x + 9 = 5;
```

expr + *expr* = *expr*;

expr = *expr*;

```
NO SUBSTITUTION
Compile Error!
```

C++ Grammar (cont.)

Rule	Substitution
statement	expr_statement | compound_statement | if (expr) statement | while (expr) statement …
compound_statement	‘{‘ statement_list ‘}’
statement_list	statement | statement_list statement

Example:

```
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) compound_statement
while(expr) statement
statement
```

Example:

```
while(x > 0)
   x--;
   x = x + 5;
```

```
while(expr)
   statement
   statement
```

```
statement
statement
```

MORE DETAILS

Recursive Functions

- Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) ***and a base/terminating case***
- Usually takes the place of a loop
- Input to the problem must be categorized as a:
  - Base case: Solution known beforehand or easily computable (no recursion needed)
  - Recursive case: Solution can be described using solutions to smaller problems of the same type
    - Keeping putting in terms of something smaller until we reach the base case
- Factorial: n! = n * (n-1) * (n-2) * … * 2 * 1
  - n! = n * (n-1)!
  - Base case: n = 1
  - Recursive case: n > 1 => n*(n-1)!

Recursive Functions (cont.)

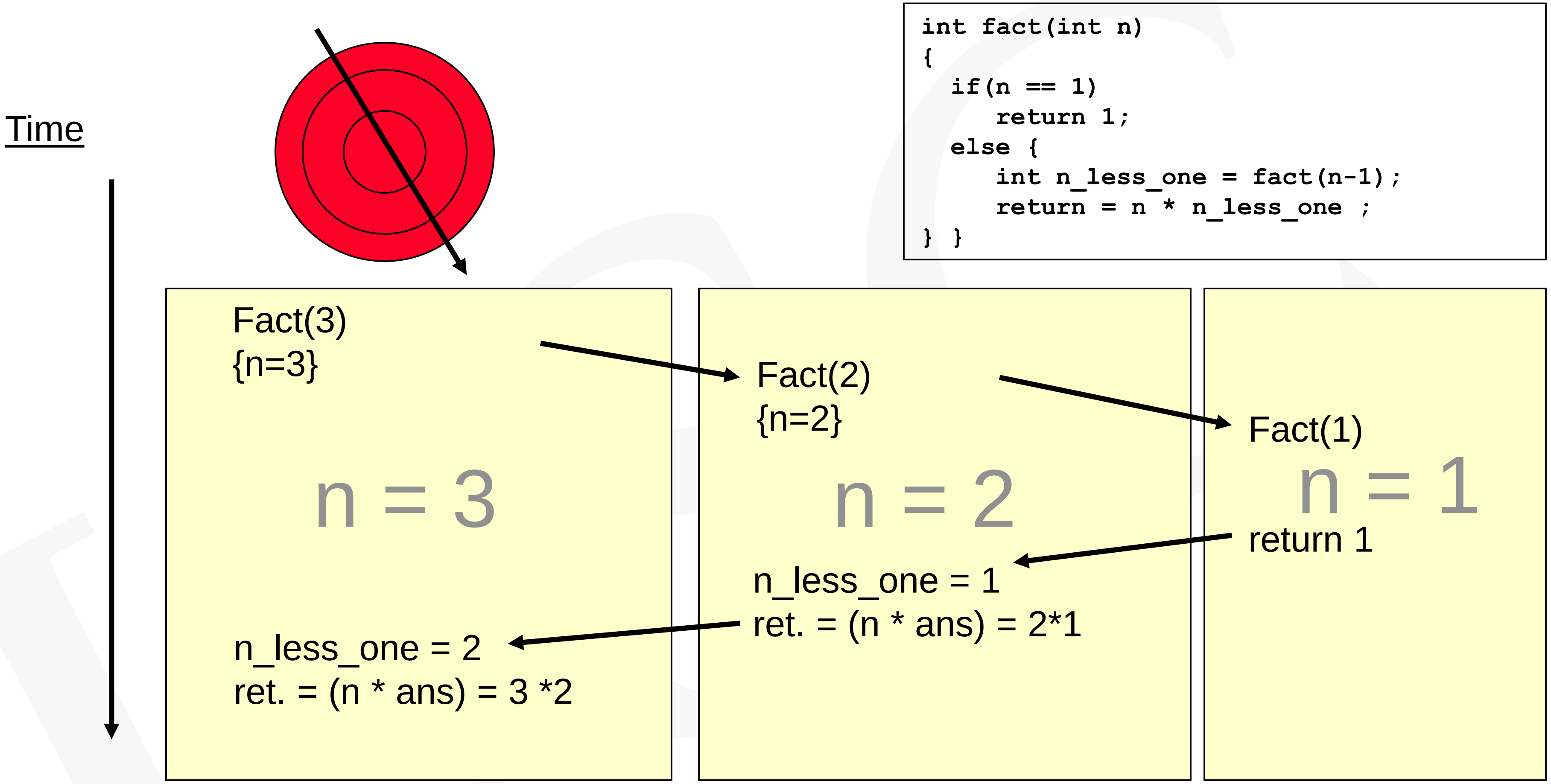
- Recall the system stack essentially provides separate areas of memory for each ‘instance’ of a function
- Thus each local variable and actual parameter of a function has its own value within that particular function instance’s memory space

C Code:

```
int fact(int n)
{
  // base case
  if(n == 1)
         return 1;
  // recursive case
  else {
     // calculate (n-1)!
     int n_less_one = fact(n-1);
        // now ans = (n-1)!
        // so calculate n!
     return = n * n_less_one ;
     }
}
```

Recursive Call Timeline

```
int fact(int n)
{
  if(n == 1)
     return 1;
  else {
     int n_less_one = fact(n-1);
     return = n * n_less_one ;
} }
```



- **Value/version of n is implicitly “saved” and “restored” as we move from one instance of the ‘fact’ function to the next**

Head vs. Tail Recursion

- **Head Recursion: Recursive call is made before the real work is performed in the function body**
- **Tail Recursion: Some work is performed and then the recursive call is made**

Tail Recursion

```
void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   cout << "Go" << endl;
   doit(n-1);
  }
}
```

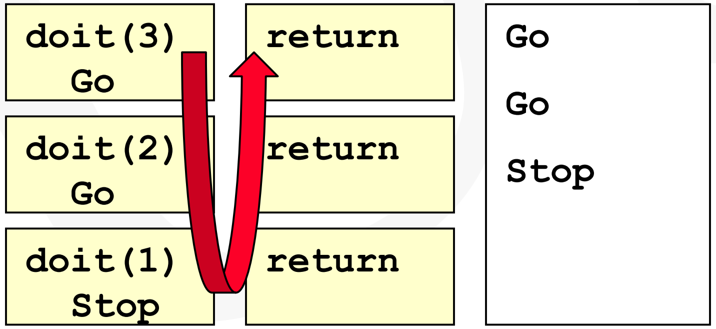
Head Recursion

```
void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   doit(n-1);
   cout << "Go" << endl;
  }
}
```

Head vs. Tail Recursion (cont.)

Tail Recursion

```
Void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   cout << "Go" << endl;
   doit(n-1);
  }
}
```



doit(3)
Go

doit(2)
Go

doit(1)
Stop

return

return

return

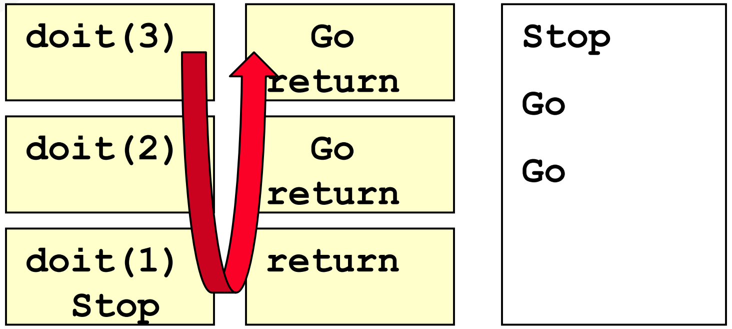
Go

Go

Stop

Head Recursion

```
Void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   doit(n-1);
   cout << "Go" << endl;
  }
}
```



doit(3)

doit(2)

doit(1)
Stop

Go
return

Go
return

return

Stop

Go

Go

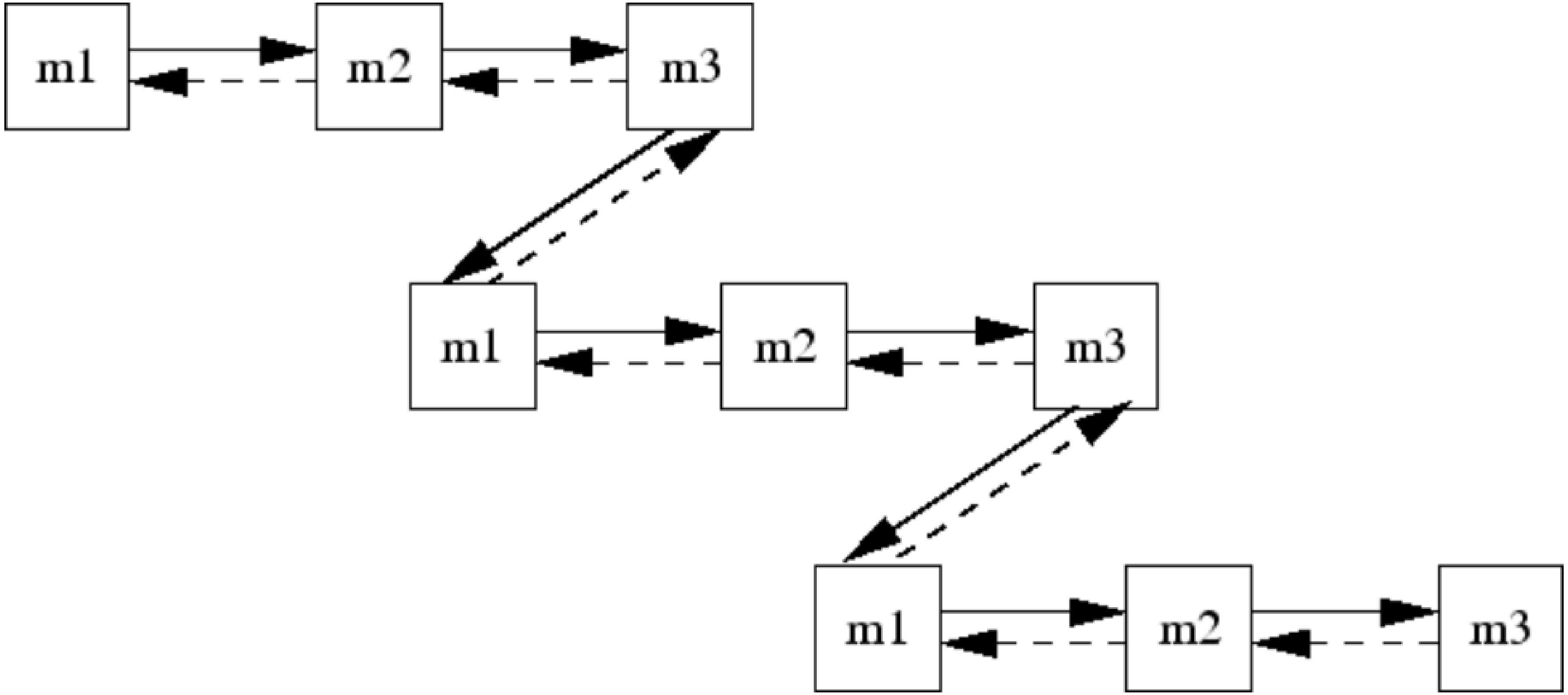
Head vs. Tail Recursion (cont.)

- Question: How would you categorize the following code?
- What would be printed?

```
void doit(int n)
{
  if(n == 1) cout << n << endl;
  else {
   cout << n << endl;
   doit(n-1);
   cout << n << endl;
  }
}
```

Direct vs. Indirect

- Occurs when a method invokes itself
- Indirect recursion occurs when a method invokes another method, eventually resulting in the original method being invoked again
- Depth of indirect recursion may vary



Recursive Functions – Example

- Recall the system stack essentially provides separate areas of memory for each ‘instance’ of a function
- Thus each local variable and actual parameter of a function has its own value within that particular function instance’s memory space

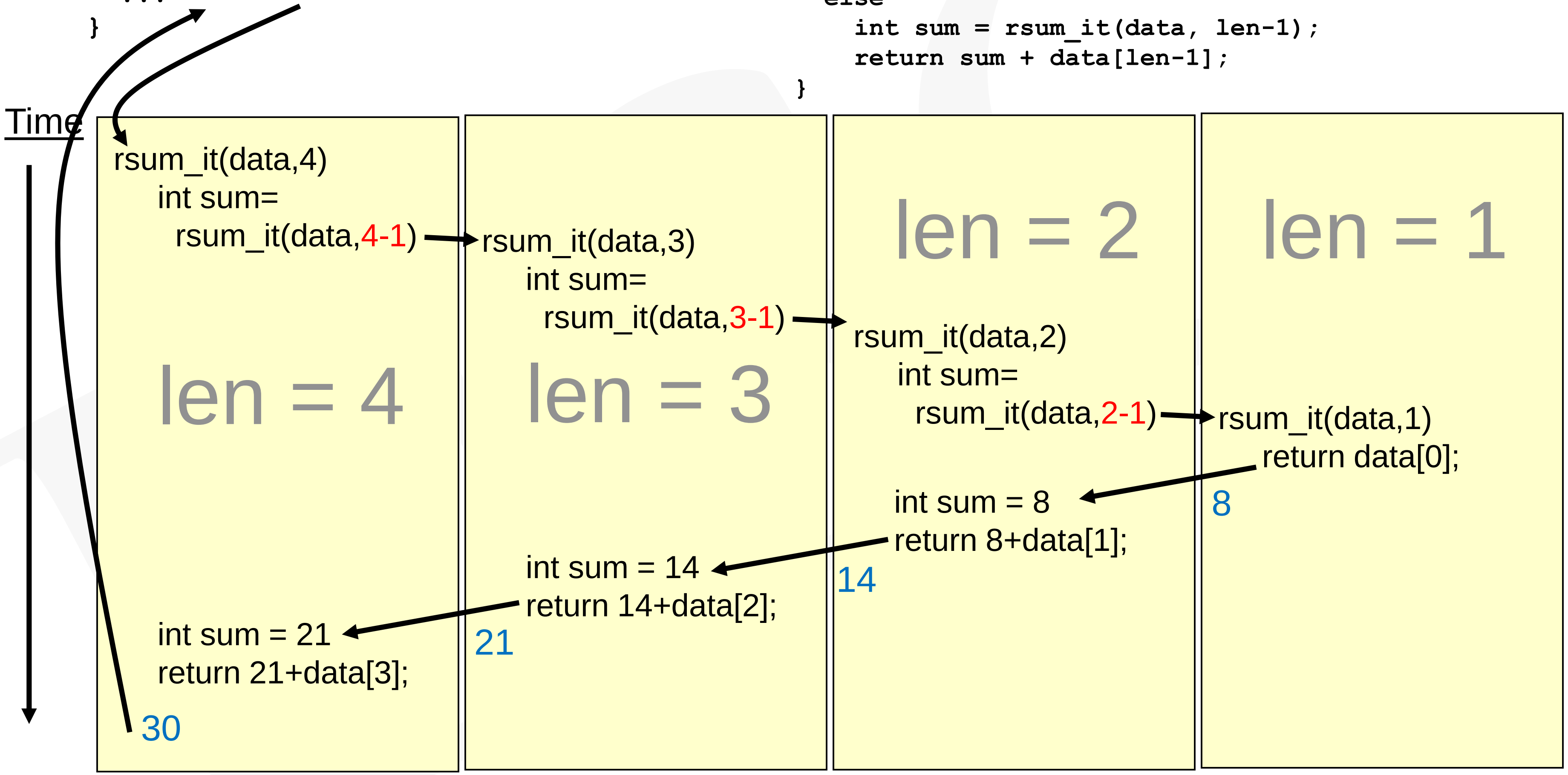
C Code:

```
int main()
{
   int data[4] = {8, 6, 7, 9};
   int sum1 = isum_it(data, 4);
   int sum2 = rsum_it(data, 4);
}

int isum_it(int data[], int len)
{
   sum = data[0];
   for(int i=1; i < len; i++){
      sum += data[i];
   }
}

int rsum_it(int data[], int len)
{
   if(len == 1)
      return data[0];
   else
      int sum = rsum_it(data, len-1);
      return sum + data[len-1];
}
```

Recursive Call Timeline

```
int main(){
  int data[4] = {8, 6, 7, 9};
  int sum2 = rsum_it(data, 4);
  ...
}
```

```
int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```



Time

len = 4

rsum_it(data,4)
int sum=
rsum_it(data,4-1)

int sum = 21
return 21+data[3];

30

len = 3

rsum_it(data,3)
int sum=
rsum_it(data,3-1)

int sum = 14
return 14+data[2];

21

len = 2

rsum_it(data,2)
int sum=
rsum_it(data,2-1)

int sum = 8
return 8+data[1];

14

len = 1

rsum_it(data,1)
return data[0];

8

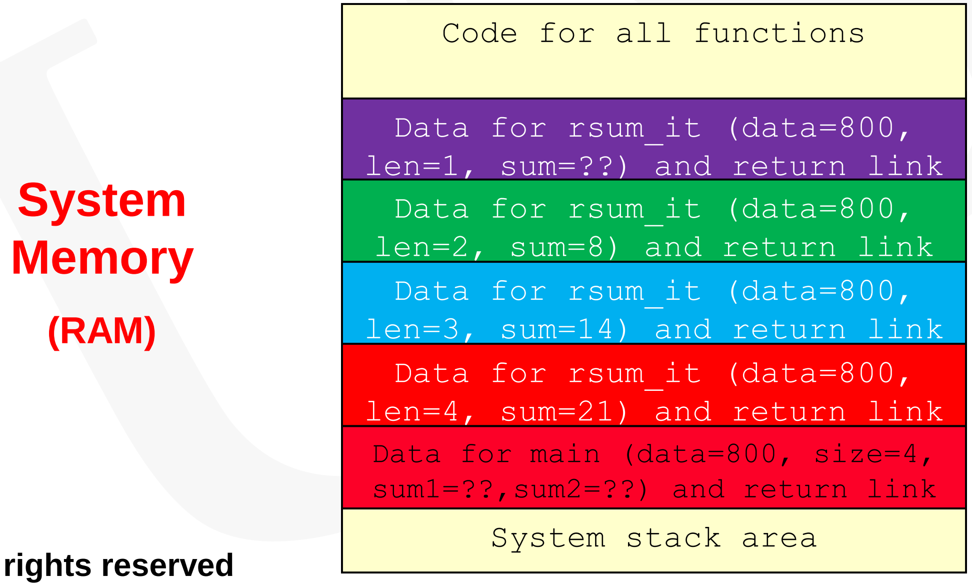
Each instance of rsum_it has its own len argument and sum variable

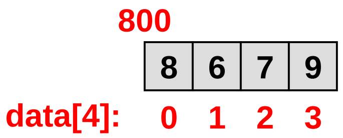
Every instance of a function has its own copy of local variables

System Stack & Recursion

- The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int sum2 = rsum_it(data, 4);
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum =
         sum_them(data, len-1);
    return sum + data[len-1];
}
```


System Memory
(RAM)
Code for all functions
Data for rsum_it (data=800, len=1, sum=??) and return link
Data for rsum_it (data=800, len=2, sum=8) and return link
Data for rsum_it (data=800, len=3, sum=14) and return link
Data for rsum_it (data=800, len=4, sum=21) and return link
Data for main (data=800, size=4, sum1=??,sum2=??) and return link
System stack area



Recursion Double Check

- When you write a recursive routine:
  - Check that you have appropriate base cases
    - Need to check for these first before recursive cases
  - Check that each recursive call makes progress toward the base case
    - Otherwise you'll get an infinite loop and stack overflow
  - Check that you use a 'return' statement at each level to return appropriate values back to each recursive call
    - You have to return back up through every level of recursion, so make sure you are returning something (the appropriate thing)

Exercise

- Count-down
- Count-up

Loops & Recursion

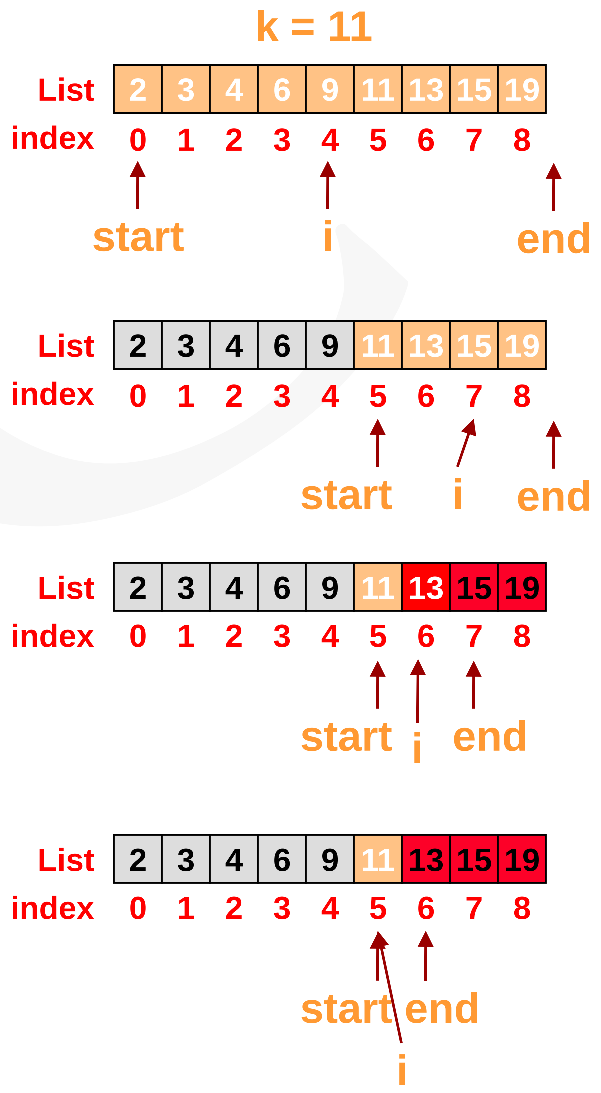
- Is it better to use recursion or iteration?
  - ANY problem that can be solved using recursion can also be solved with iteration and other appropriate data structures
- Why use recursion?
  - Usually clean & elegant. Easier to read
  - Sometimes generates much simpler code than iteration would
  - Sometimes iteration will be almost impossible
- How do you choose?
  - Iteration is usually faster and uses less memory
  - However, if iteration produces a very complex solution, consider recursion

Exercise

- Exercises
  - Text-based fractal

Recursive Binary Search

- Assume remaining items = [start, end)
  - start is inclusive index of start item in remaining list
  - End is exclusive index of start item in remaining list
- binSearch(target, List[], start, end)
  - Perform base check (empty list)
    - Return NOT FOUND (-1)
  - Pick mid item
  - Based on comparison of k with List[mid]
    - EQ => Found => return mid
    - LT => return answer to BinSearch[start,mid)
    - GT => return answer to BinSearch[mid+1,end)



k = 11

List	2	3	4	6	9	11	13	15	19
index	0	1	2	3	4	5	6	7	8

start i end

List	2	3	4	6	9	11	13	15	19
index	0	1	2	3	4	5	6	7	8

start i end

List	2	3	4	6	9	11	13	15	19
index	0	1	2	3	4	5	6	7	8

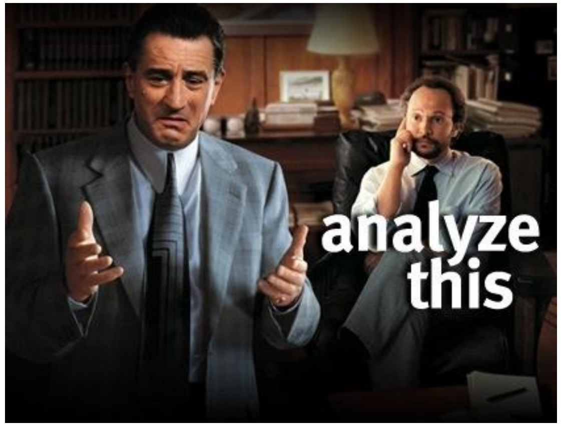
start i end

List	2	3	4	6	9	11	13	15	19
index	0	1	2	3	4	5	6	7	8

start end

i

Analyze These!

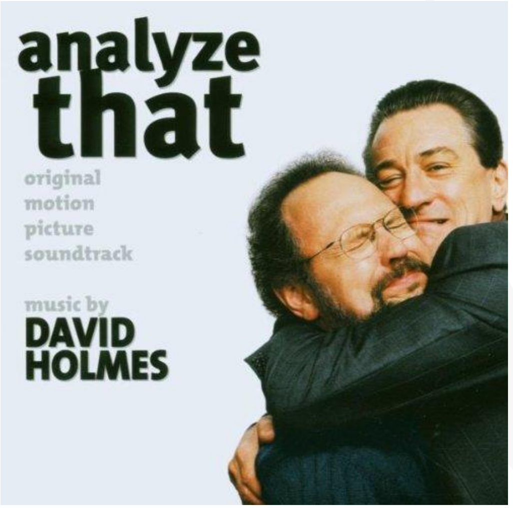


- What does this function print?

```
void rfunc(int n, int t) {
   if (n == 0) {
      cout << t << " ";
      return;
   }
   rfunc(n-1, 3*t);
   rfunc(n-1, 3*t+2);
   rfunc(n-1, 3*t+1);
}
int main() {
   rfunc(2, 0);
}
```

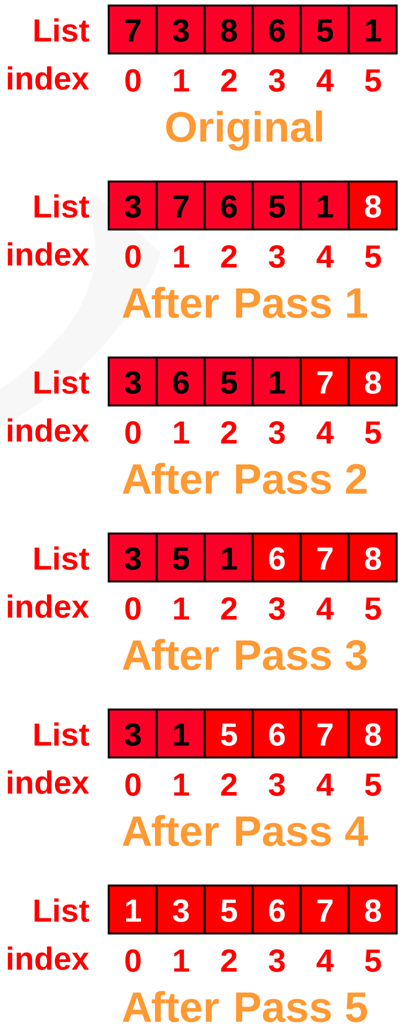
- What does this function return for g(3122013)

```
int g(int n) {
   if (n % 2 == 0)
      return n/10;
   return g(g(n/10));
}
```



Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity, i.e., faster or slower
- Bubble Sort (simple though not terribly efficient)
  - On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e., w/o the newly placed maximum value)



List	7	3	8	6	5	1
index	0	1	2	3	4	5

Original

List	3	7	6	5	1	8
index	0	1	2	3	4	5

After Pass 1

List	3	6	5	1	7	8
index	0	1	2	3	4	5

After Pass 2

List	3	5	1	6	7	8
index	0	1	2	3	4	5

After Pass 3

List	3	1	5	6	7	8
index	0	1	2	3	4	5

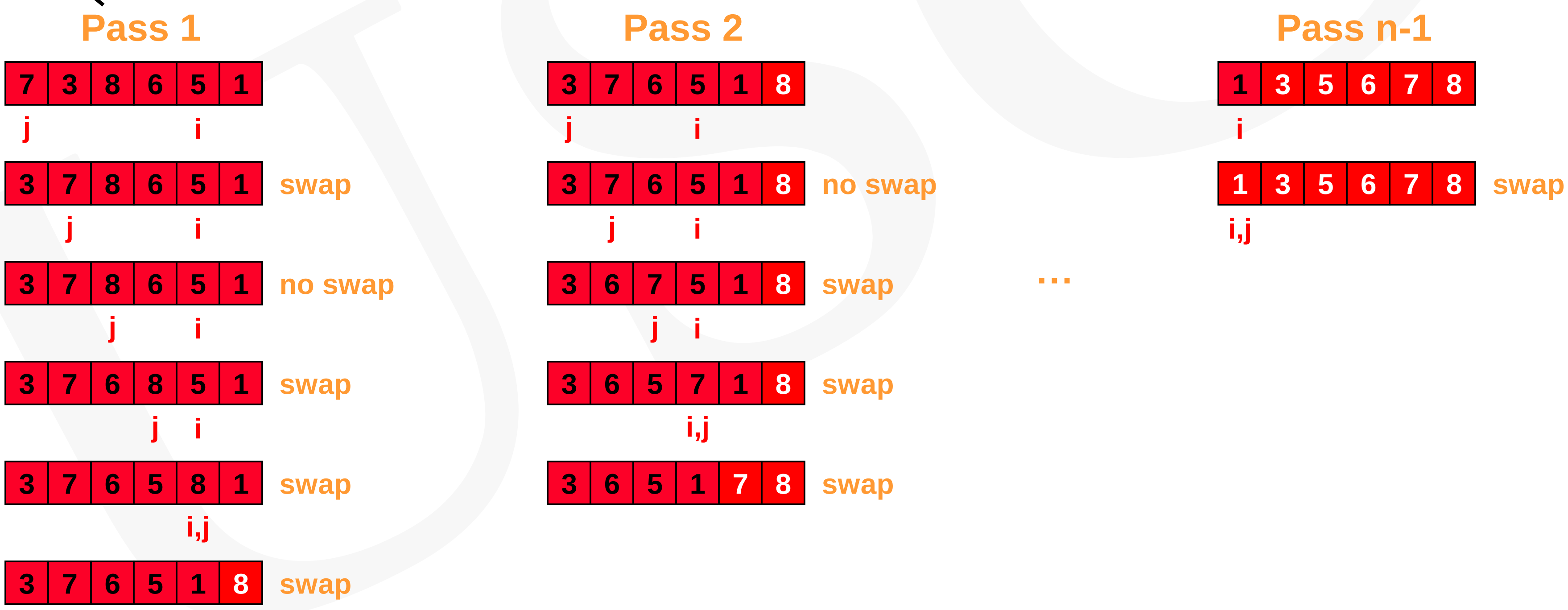
After Pass 4

List	1	3	5	6	7	8
index	0	1	2	3	4	5

After Pass 5

Bubble Sort Algorithm

```
n ← length(List);
for( i=n-2; i >= 1; i--)
  for( j =1; j <= i; j++)
    if ( List[j] > List[j+1] ) then
      swap List[j] and List[j+1]
```



Pass 1

7 3 8 6 5 1
j i

3 7 8 6 5 1 swap
j i

3 7 8 6 5 1 no swap
j i

3 7 6 8 5 1 swap
j i

3 7 6 5 8 1 swap
i,j

3 7 6 5 1 8 swap

Pass 2

3 7 6 5 1 8
j i

3 7 6 5 1 8 no swap
j i

3 6 7 5 1 8 swap
j i

3 6 5 7 1 8 swap
i,j

3 6 5 1 7 8 swap

...

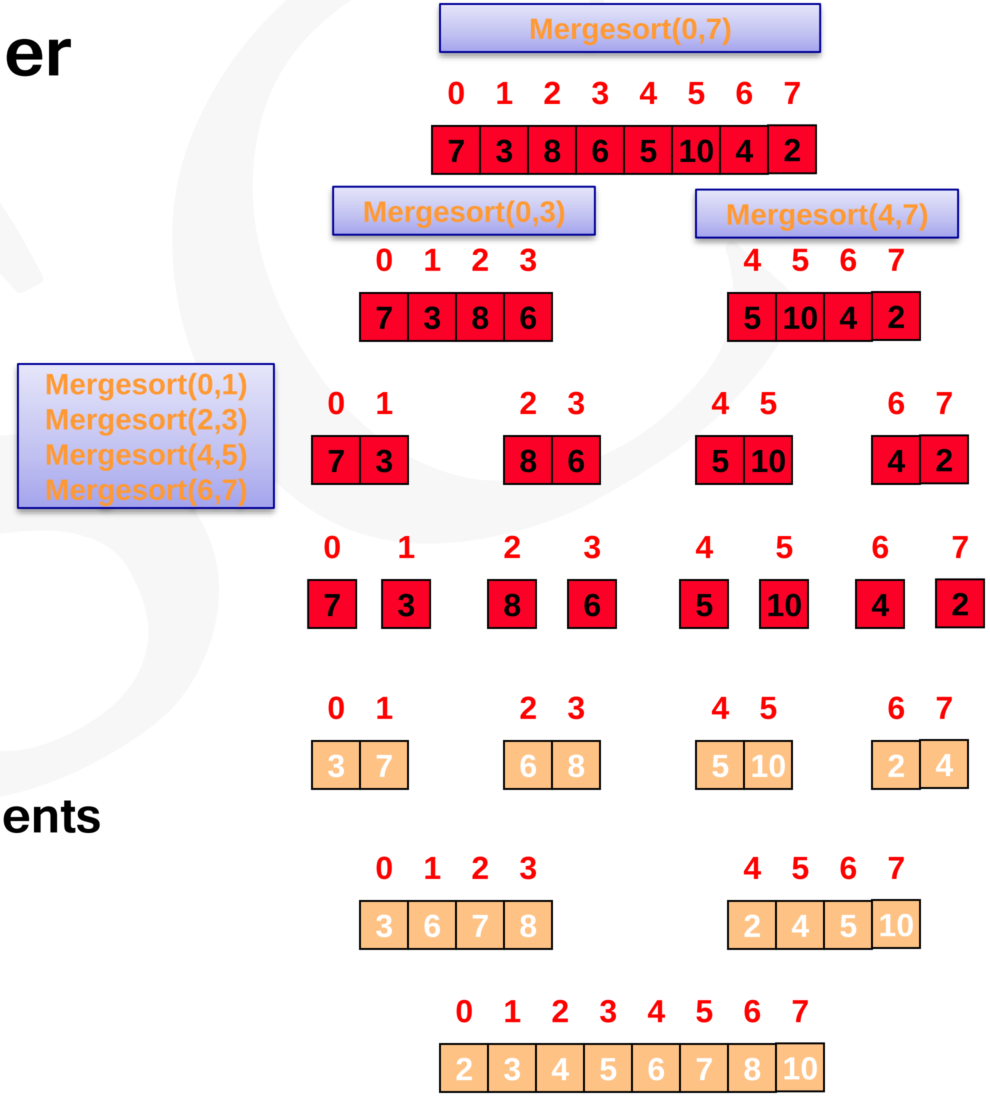
Pass n-1

1 3 5 6 7 8
i

1 3 5 6 7 8 swap
i,j

MergeSort – Recursive Sort

- Break sorting problem into smaller sorting problems and merge the results at the end
- Mergesort(0..n-1)
  - If list is size 1, return
  - Else
    - Mergesort(0..n/2)
    - Mergesort(n/2+1 .. n-1)
    - Combine each sorted list of n/2 elements into a sorted n-element list



Mergesort(0,7)

0 1 2 3 4 5 6 7

7 3 8 6 5 10 4 2

Mergesort(0,3)

0 1 2 3

7 3 8 6

Mergesort(4,7)

4 5 6 7

5 10 4 2

Mergesort(0,1)
Mergesort(2,3)
Mergesort(4,5)
Mergesort(6,7)

0 1 | 2 3 | 4 5 | 6 7

7 3 | 8 6 | 5 10 | 4 2

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

7 | 3 | 8 | 6 | 5 | 10 | 4 | 2

0 1 | 2 3 | 4 5 | 6 7

3 7 | 6 8 | 5 10 | 2 4

0 1 2 3 | 4 5 6 7

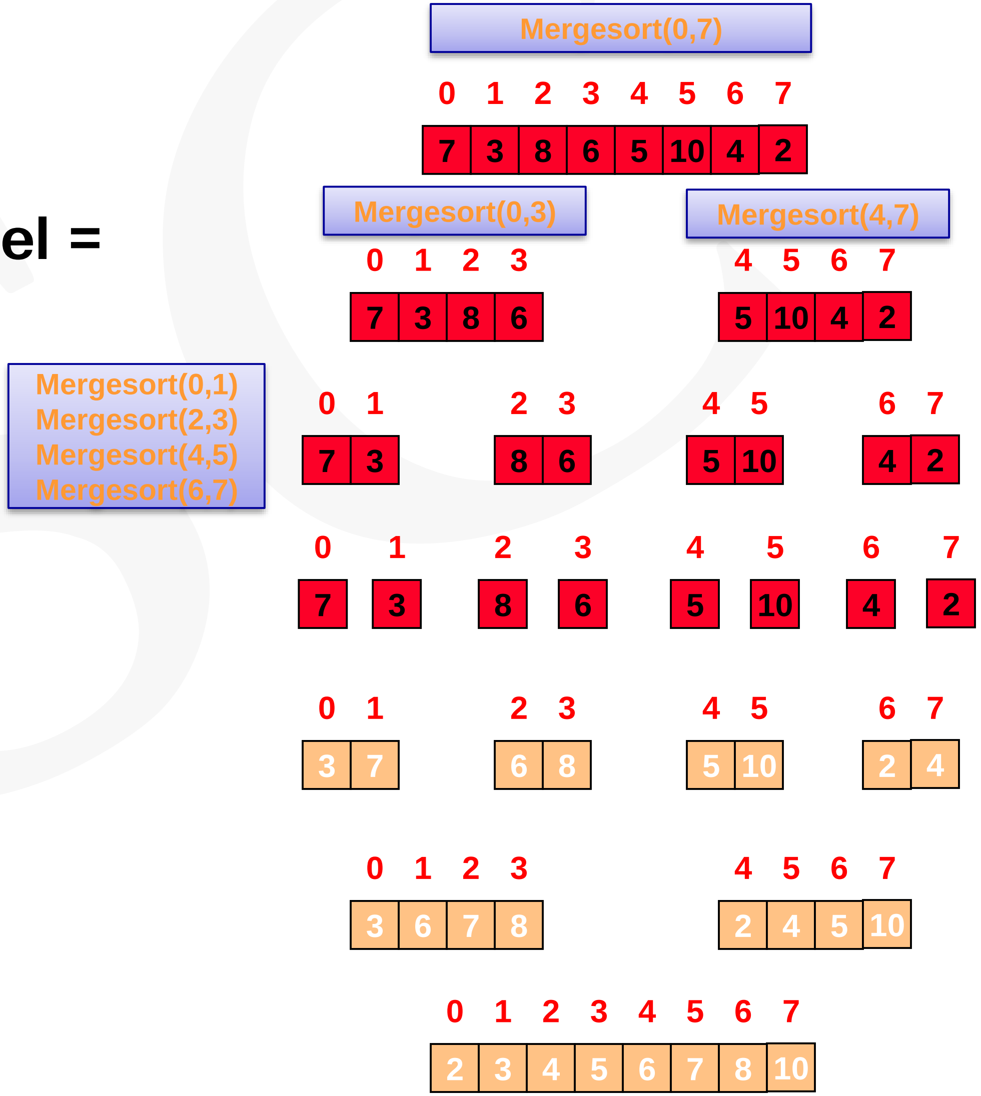
3 6 7 8 | 2 4 5 10

0 1 2 3 4 5 6 7

2 3 4 5 6 7 8 10

MergeSort – Recursive Sort (cont.)

- Run-time analysis
  - # of recursion levels =
    - $Log_2(n)$
  - Total operations to merge each level =
    - n operations total to merge two lists over all recursive calls
- Mergesort = O(n * lg(n))
  - lg(n) is shorthand for $log_2(n)$ [i.e. log base 2]



Another Example

- Shown at the right are the binary combinations for different numbers of bits
- Do you see a recursive pattern of the combinations as you look at progressively larger numbers of bits?
  - Hint: Start at the leftmost bit and move rightward

1-bit Bin.	2-bit Bin.	3-bit Bin.	4-bit Bin.
0	00	000	0000
1	01	001	0001
	10	010	0010
	11	011	0011
		100	0100
		101	0101
		110	0110
		111	0111
			1000
			1001
			1010
			1011
			1100
			1101
			1110
			1111

Another Example (cont.)

- If you are given the value, n, and an array with n characters could you generate all the combinations of n-bit binary?
- Do so recursively!

binarynumbers.cpp

```
0
1
```

1-bit Bin.

```
00
01
10
11
```

2-bit Bin.

```
000
001
010
011
100
101
110
111
```

3-bit Bin.

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

4-bit Bin.

Examples

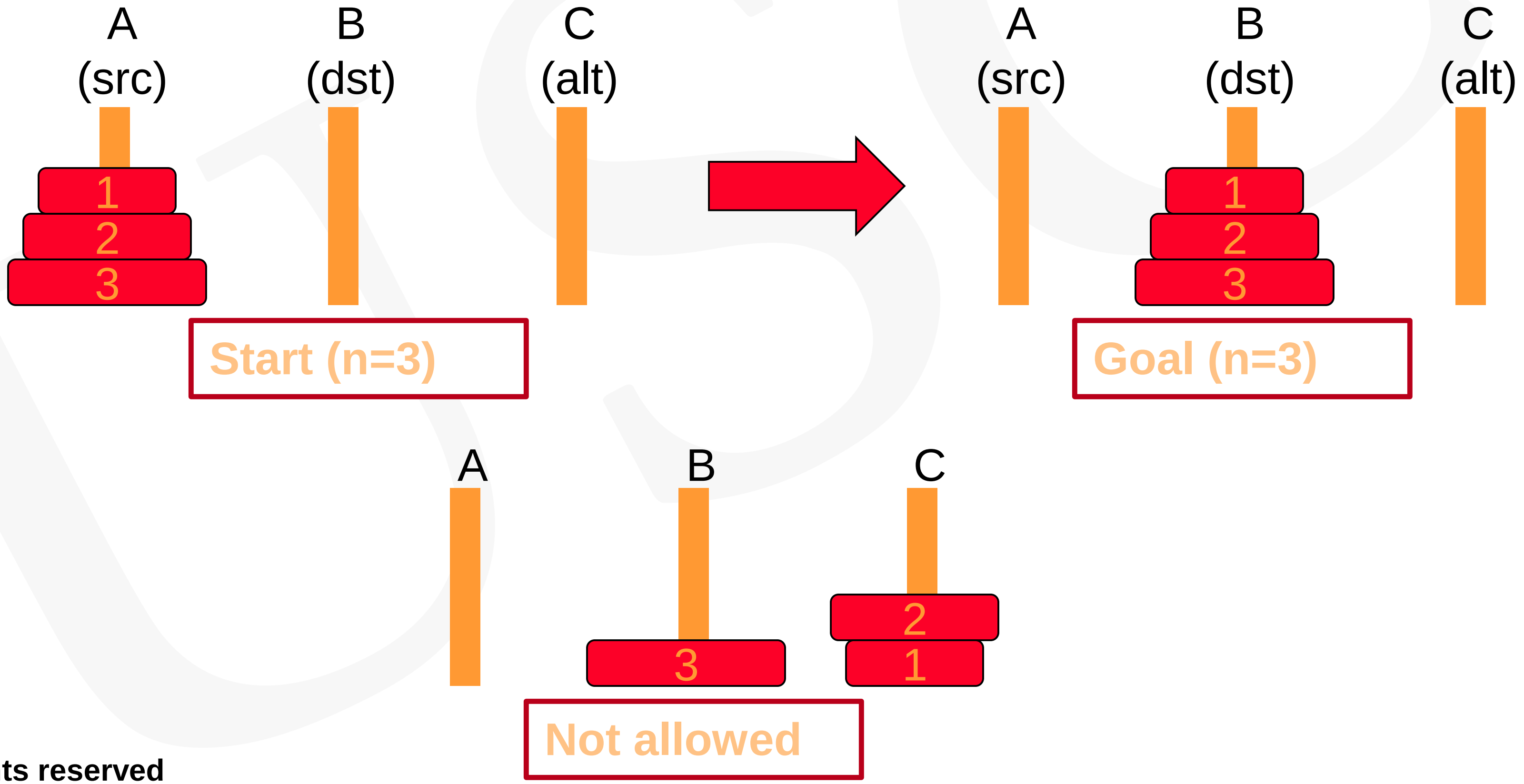
- In-class-exercises
  - Zero_sum
  - Basen_combos

bitcompos.cpp

OTHER RECURSIVE EXAMPLES

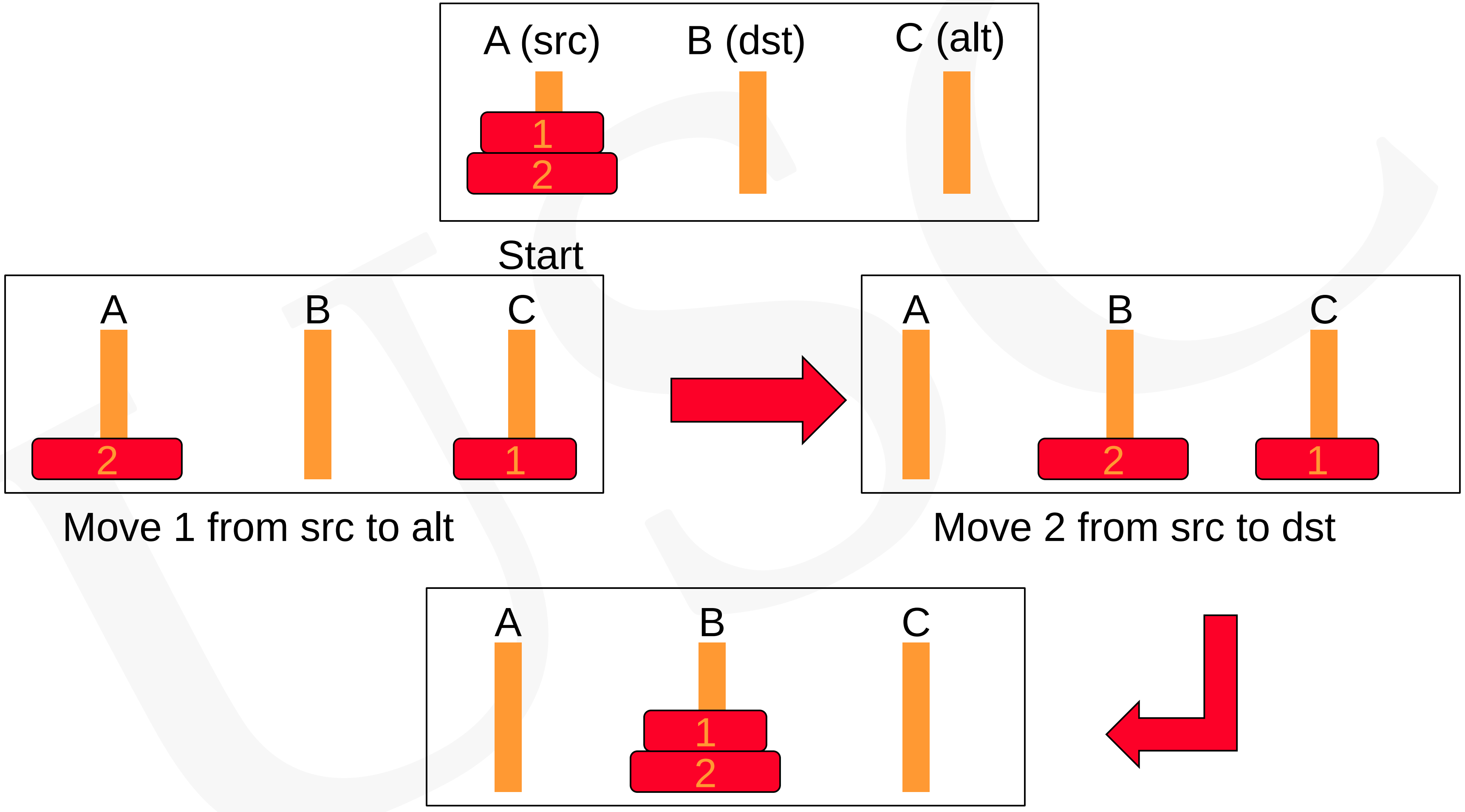
Towers of Hanoi Problem

- **Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)**
  - **Cannot place a larger disc on top of a smaller disc**
  - **Can only move one disc at a time**



Observation 1

- **Observation 1: Disc 1 (smallest) can always be moved**
- **Solve the n=2 case:**



Start

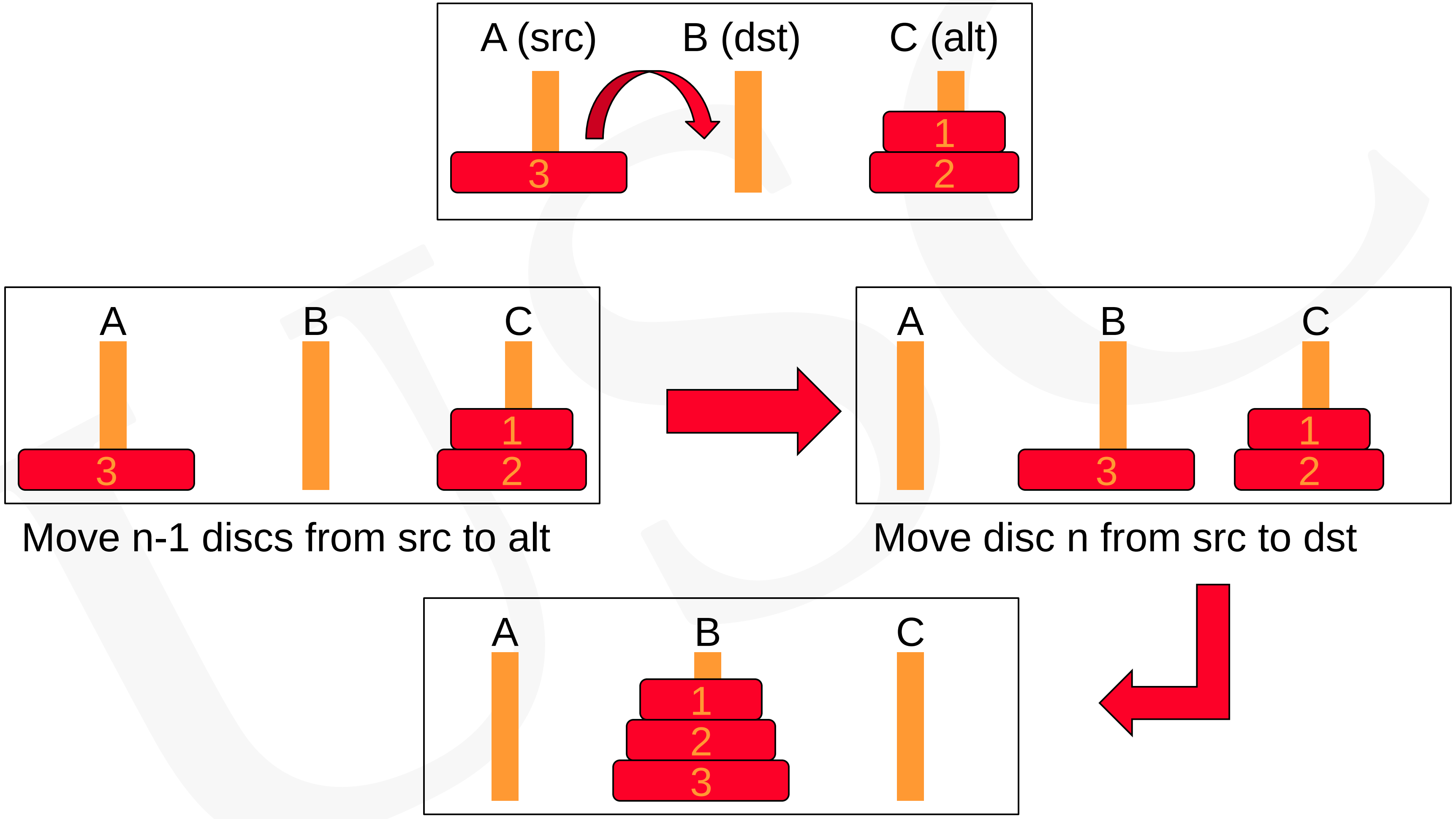
Move 1 from src to alt

Move 2 from src to dst

Move 1 from alt to dst

Observation 2

- **Observation 2: If there is only one disc on the src pole and the dest pole can receive it the problem is trivial**



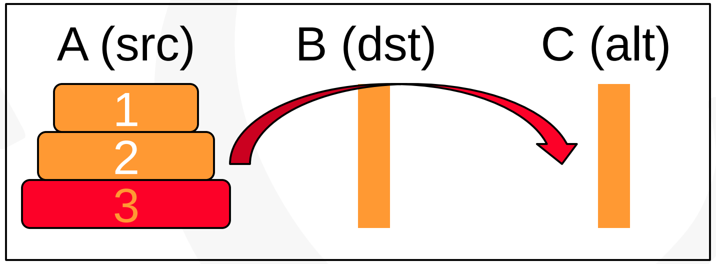
Move n-1 discs from src to alt

Move disc n from src to dst

Move n-1 discs from alt to dst

Recursive solution

- But to move n-1 discs from src to alt is really a smaller version of the same problem with
  - n => n-1
  - src=>src
  - alt =>dst
  - dst=>alt
- Towers(n,src,dst,alt)
  - Base Case: n==1 // Observation 1: Disc 1 always movable
    - Move disc 1 from src to dst
  - Recursive Case: // Observation 2: Move of n-1 discs to alt & back
    - Towers(n-1,src,alt,dst)
    - Move disc n from src to dst
    - Towers(n-1,alt,dst,src)



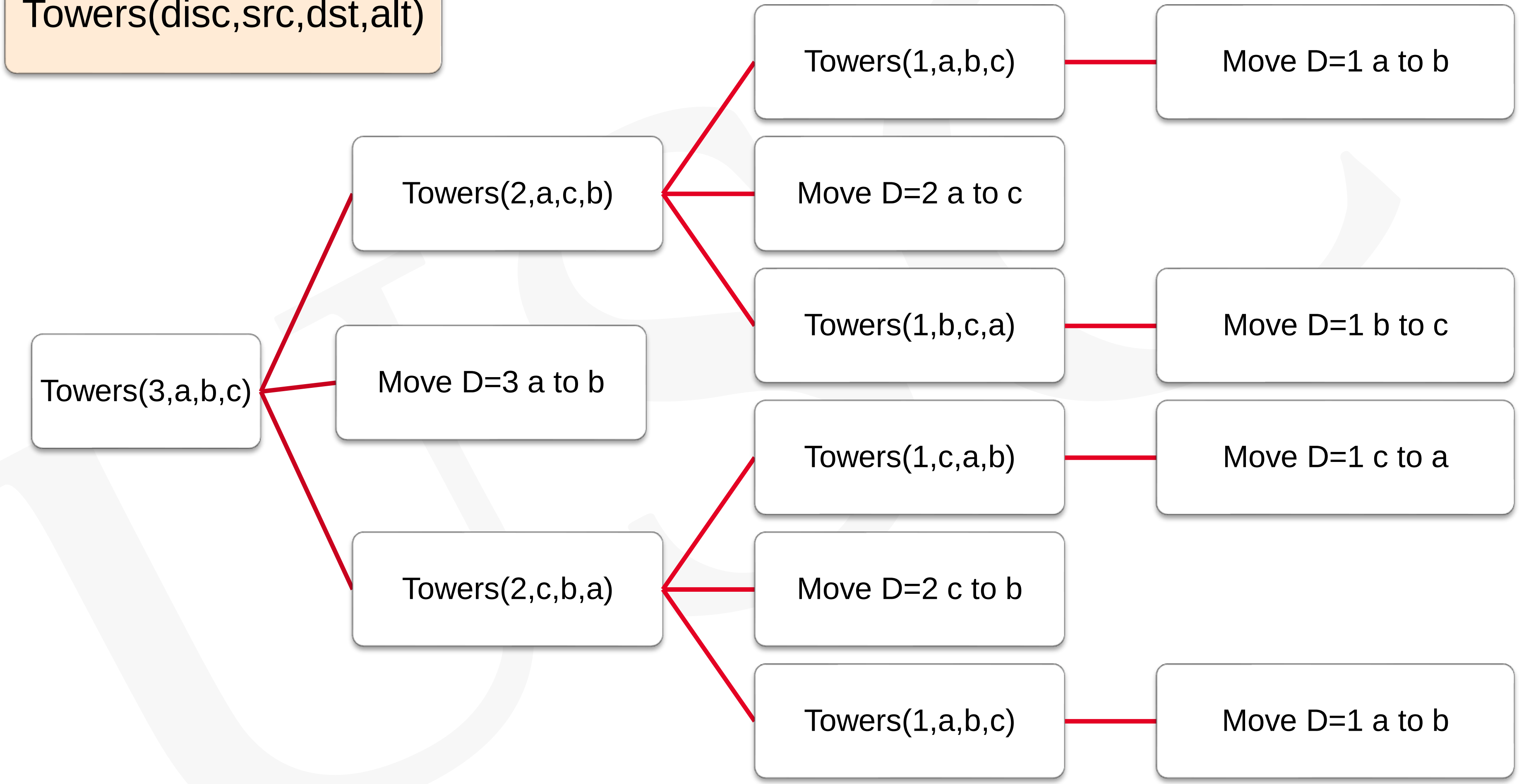
Exercise

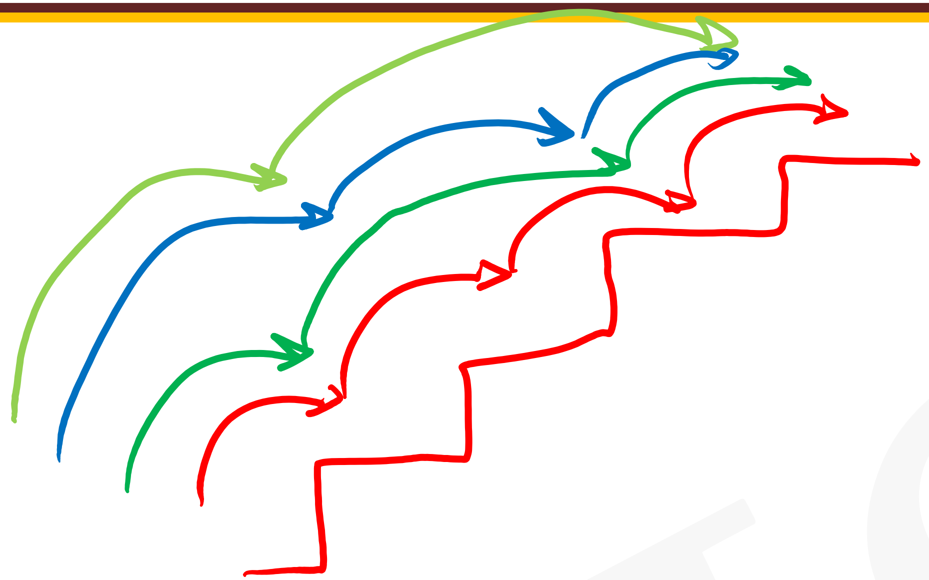
- Implement the Towers of Hanoi code
  - hanoi.cpp
  - Just print out "move disc=x from y to z" rather than trying to "move" data values
    - Move disc 1 from a to b
    - Move disc 2 from a to c
    - Move disc 1 from b to c
    - Move disc 3 from a to b
    - Move disc 1 from c to a
    - Move disc 2 from c to b
    - Move disc 1 from a to b

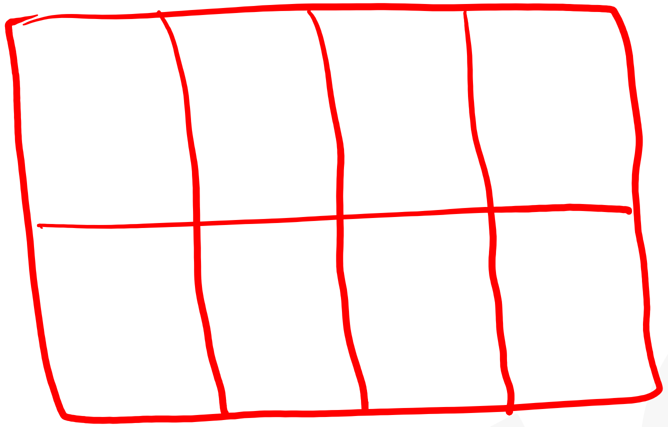
Recursive Box Diagram

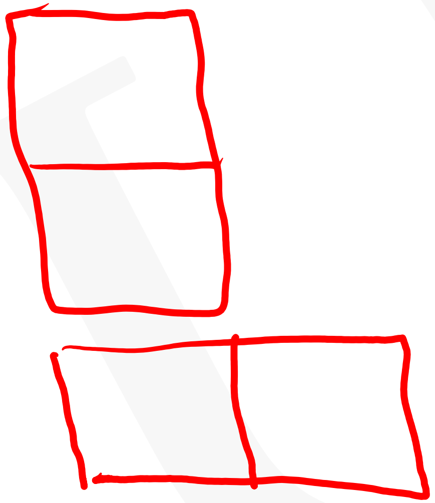
Towers Function Prototype

Towers(disc,src,dst,alt)









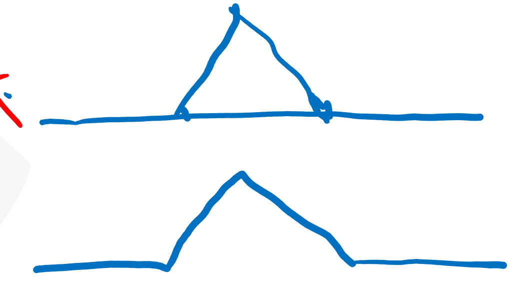
1- Start a line

2- Divide it into 3 equal segments

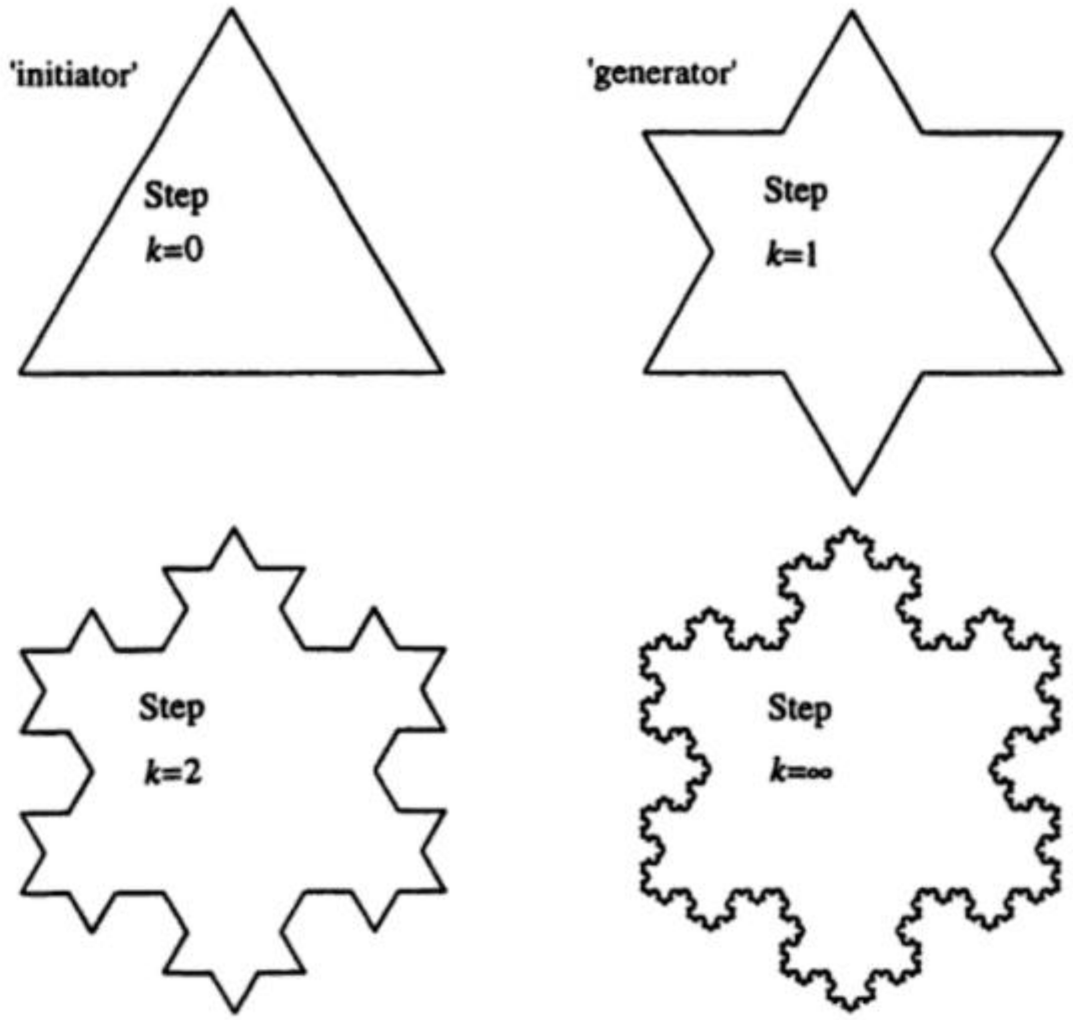
3- Draw an equilateral Δ on the middle segment

4- erase the base of Δ

5- repeat 2-4 for the remaining lines again and again



Version 2



More Versions