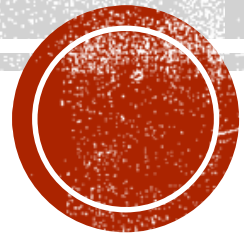
INTRODUCTION



Ekim 2013

A GOOD PROGRAM

- Meet requirements
- Run correctly
- Run efficiently
- Be easy to read and understand
- Be easy to debug
- Be easy to modify
- Be documented

PROGRAM

Program = Algorithm + Data Structures

Algorithm : method for solving a problem

Data Structure : method to store information

DATA STRUCTURE

Data structure is a specialized format for organizing and storing data so that it can be accessed and worked with in appropriate ways to make a program efficient.

- Arrays
- Stacks
- Queues
- Trees
- Lists
- Graphs

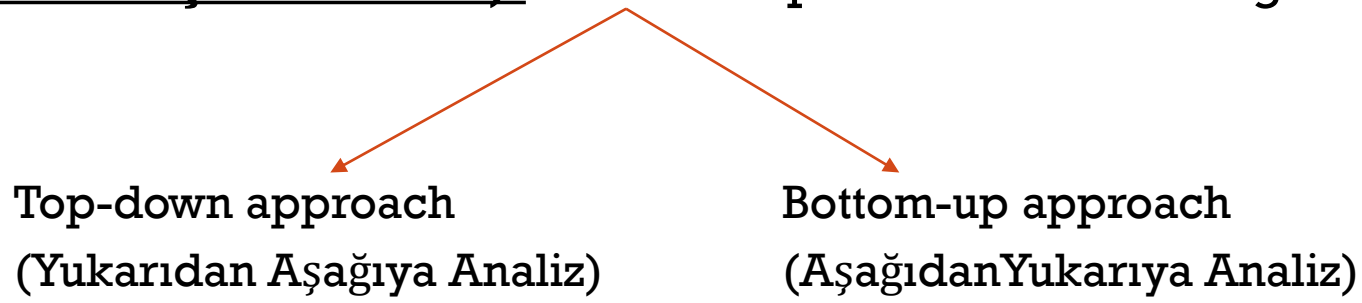
SYSTEM LIFE CYCLE

1. **Requirements (Gereksinimler):** Initial specifications are defined.

   input and output descriptions.

   what it does.

2. **Analysis (Analiz-Çözümleme):** break the problem into manageable pieces.

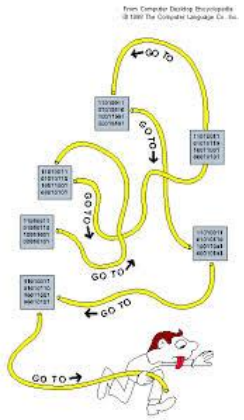


SYSTEM LIFE CYCLE

3. **Design(Tasarım):**
   the abstract data types, the algorithm specifications.
   language-independent.
   create a system that could be written in several PL.

4. **Refinement and Coding (Arıtma ve Kodlama):**
   choose representations for our data objects and write algorithms.

5. **Verification (Doğrulama)**
   correctness proofs
   testing
   error removal



ALGORITHM

A finite set of instructions which accomplishes a particular task.

- Input
- Output
- Definiteness: each instruction is clear and unambigious.
- Finiteness
- Effectiveness

SELECTION SORT

A program that sorts a set of n>=1 integers.

```
for(i = 0 ; i < n; i++)
{
        examine list[i] to list[n-1] and suppose that the smallest
    integer is at list[min];
        interchange list[i] and list[min]
}
```

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, z) (z = x; x = y; y = z)
void sort(int [], int);

int main()
{
        int i, n;
        int list[MAX_SIZE];
        printf("dizi boyu");
        scanf("%d", &n);
        //n kontrol et..
        for(i = 0; i < n; i++){
                list[i] = rand() % 100;
                printf("%d ", list[i]);
        }
        sort(list,n);
        //sıralı değerler yazdır..
        return 0;
}
```

```
void sort(int list[], int n)
{
        int i, j, min, temp;
        for(i = 0; i < n-1; i++){
          min = i;
          for(j = i+1; j < n; j++)
                if(list[j] < list[i])
                  min = j;
          SWAP(list[i], list[min], temp);
        }
}
```

SWAP

```
swap(&list[i], &list[min]);


void swap(int *x, int *y)
{
        int temp = *x;
        *x = *y ;
        *y = temp;
}
```

BINARY SEARCH

Check if an integer is in an ordered list given. If it is not present -1, otherwise the index of the element in the list.

```
while (there are more integers to check){
        middle = (left + right) / 2;
        if(searchnum < list[middle])
                right = middle – 1;
        else if(searchnum == list[middle])
                return middle;
        else
                left = middle+1;
}
```

```
int compare(int x, int y)
{
      if(x < y) return -1;
      else if(x > y) return 1;
      else return 0;
}
```

```
int binsearch(int list[], int searchnum, int left, int right)
{
      int middle;
      while(left <= right){
          middle = (left + right)/2;
             switch(compare(list[middle], searchnum)){
                    case -1 : left = middle + 1; break;
                    case 1: rigt = middle – 1; break;
                    case 0;  return middle;
             }
       }
       return -1 ;
}
```

BINARY SEARCH *EX*

4	13	18	19	24	27	33	36	39	40
LEFT									RIGHT

4	13	18	19	24	27	33	36	39	40
LEFT			RIGHT						

4	13	18	19	24	27	33	36	39	40
		LEFT	RIGHT						

This is the iterative version of the Binary Search algorithm. We could write the recursive one!!

RECURSION

Recursion is the process whereby a construct operates on itself.

In C, a function may directly or indirectly call itself in the course of execution.

- **direct** the call to a function occurs inside the function itself
- **indirect** a function calls another function, which in turn makes a call to the first one

Recursion is a programming technique that naturally implements the divide-and-conquer problem solving methodology.

THE NATURE OF (DIRECT) RECURSION

1. One or more simple cases of the problem (called the *stopping cases or base case*) have a simple non-recursive solution.
2. The other cases of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.
3. Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

In general:

if *(stopping case)*

solve it

else

reduce the problem using recursion

AN EXAMPLE - PALINDROME

- A **palindrome** is a word, phrase, number or other sequence of units that can be read the same way in either direction.

  e.g., *radar, level, rotator,* "Step on no pets",
  "Ey Edip Adanada pide ye", etc.

Recursive definition:

If the string has no elements, then it's a **palindrome.**

If the string has only one element, then it's a **palindrome.**

If the elements in the endpoints (the first and last elements) are the same, and the internal letters are a **palindrome,** then it's a **palindrome.**

FOUR CRITERIA OF A RECURSIVE SOLUTION

1. A recursive function calls itself.
   - This action is what makes the solution recursive.

2. Each recursive call solves an identical, but smaller, problem.
   - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.

3. A test for the base case enables the recursive calls to stop.
   - There must be a case of the problem (known as *base case* or *stopping case*) that is handled differently from the other cases (without recursively calling itself.)
   - In the base case, the recursive calls stop and the problem is solved directly.

4. Eventually, one of the smaller problems must be the base case.
   - The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.

FACTORIAL FUNCTION : ITERATIVE DEFINITION

n! = n * (n-1) * (n-2) * … * 2 * 1 for any integer n>0

0! = 1

Iterative Definition in C:

```
fval = 1;
for (i = n; i >= 1; i--)
   fval = fval * i;
```

FACTORIAL FUNCTION : RECURSIVE DEFINITION

- To define ***n!*** recursively, ***n!*** must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):

  n! = n * (n-1)!
- Base case: 0! = 1
- We can reach the base case, by subtracting 1 from n if n is a positive integer.

Recursive Definition:

$n! = 1$ if $n = 0$

$n! = n*(n-1)!$ if $n > 0$

FACTORIAL FUNCTION : RECURSIVE DEFINITION

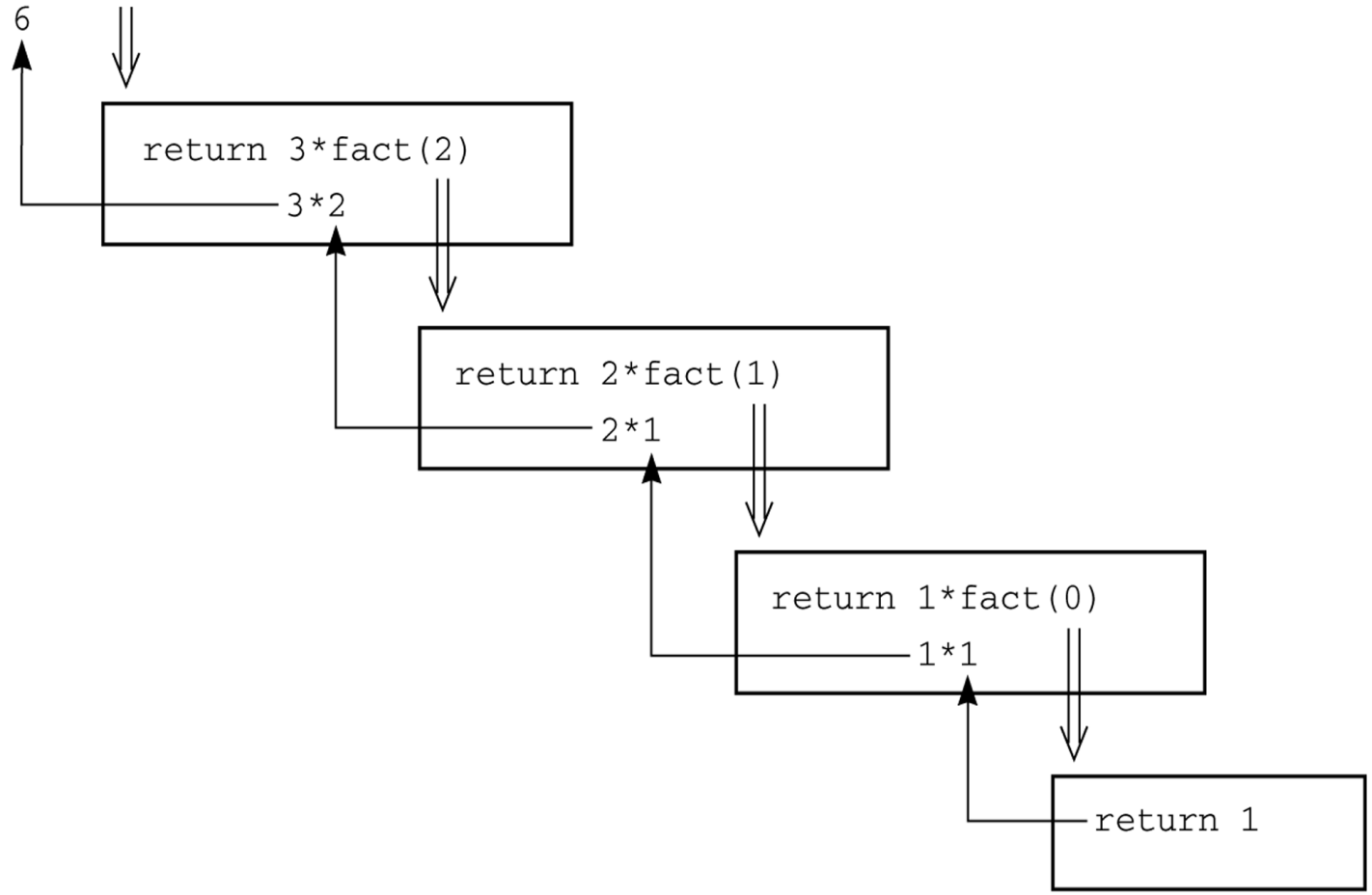
```
/* Computes the factorial of a nonnegative integer.
   Precondition: n must be greater than or equal to 0.
   Postcondition: Returns the factorial of n; n is unchanged.*/

int fact(int n)
{
   if (n ==0)
        return (1);
   else
        return (n * fact(n-1));
}
```

This ***fact*** function satisfies the four criteria of a recursive solution.

A SEQUENCE OF COMPUTATIONS OF FACTORIAL FUNCTION

```
printf("%d", fact (3));
```



TRACING A RECURSIVE FUNCTION

- A **stack** is used to keep track of function calls.
- Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)
  - For each function call, an ***activation record*** is created on the stack.
- To trace a recursive function, the ***box method*** can be used.
  - The box method is a systematic way to trace the actions of a recursive function.
  - The box method illustrates how compilers implement recursion.
  - Each box in the box method roughly corresponds to an activation record.

THE BOX METHOD (FOR A VALUED FUNCTION)

1. Label each recursive call in the body of the recursive function.
   - These labels help us to keep track of the correct place to which we must return after a function call completes.
   - After each recursive call, we return to the labeled location, and substitute that recursive call with returned valued.

```
if (n ==0)
    return (1);
else
  return (n * fact(n-1) )
```

A

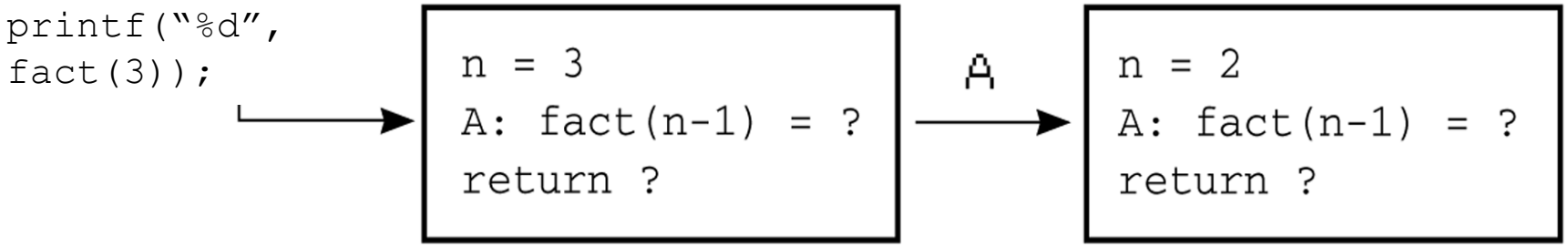
THE BOX METHOD (CONTINUED)

2. Each time a function is called, a new box represents its local environment. Each box contains:
   - the values of the arguments,
   - the function local variables
   - A placeholder for the value returned from each recursive call from the current box. (label in step 1)
   - The value of the function itself.

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

THE BOX METHOD (CONTINUED)

3. Draw an arrow from the statement that initiates the recursive process to the first box.
   - Then draw an arrow to a new box created after a recursive call, put a label on that arrow.



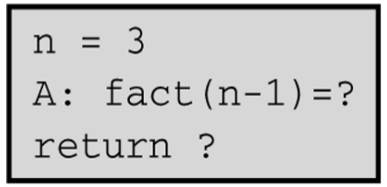
THE BOX METHOD (CONTINUED)

4. After a new box is created, we start to execute the body of the function.

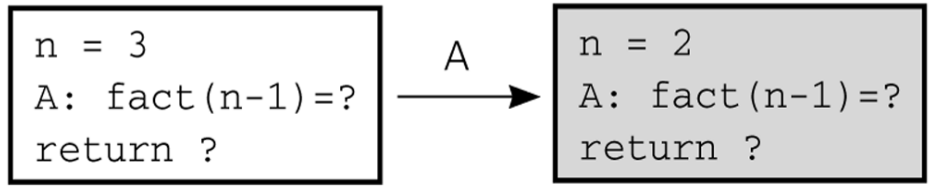
5. On exiting a function, cross off the current box an follow its arrow back to the box that called the function.
   - This box becomes the current box.
   - Substitute the value returned by the just-terminated function call into the appropriate item in the current box.
   - Continue the execution from the returned point.

BOX TRACE OF FACT(3)

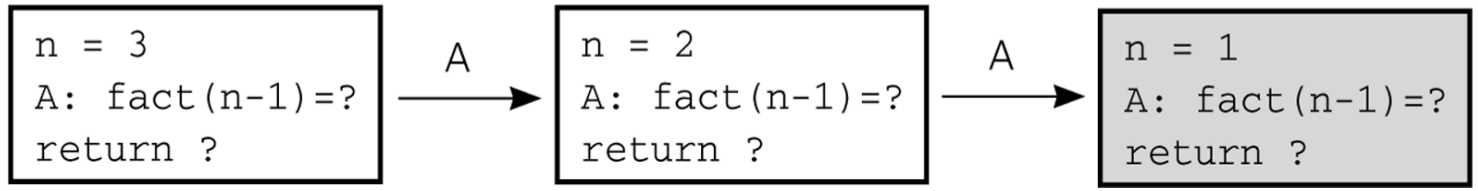
The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

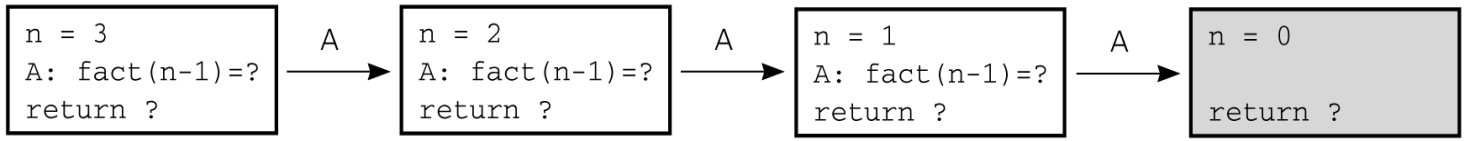


At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

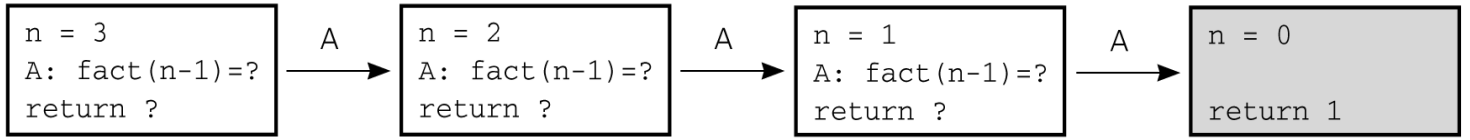


BOX TRACE OF FACT(3)

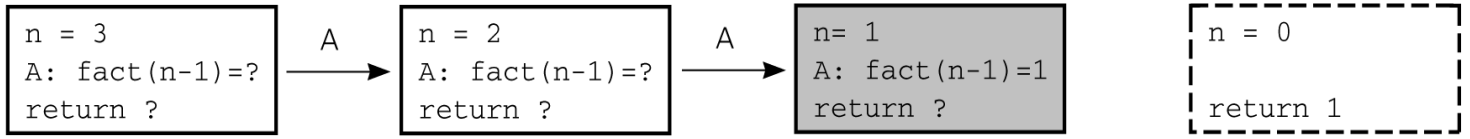
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



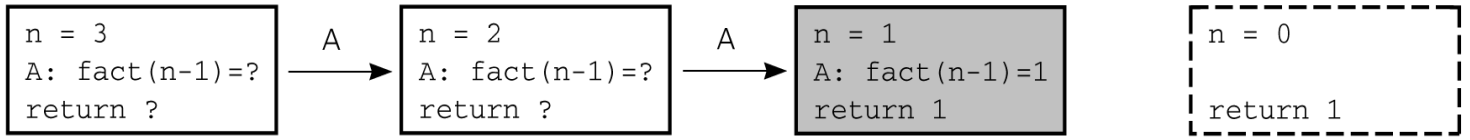
This is the base case, so this invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:

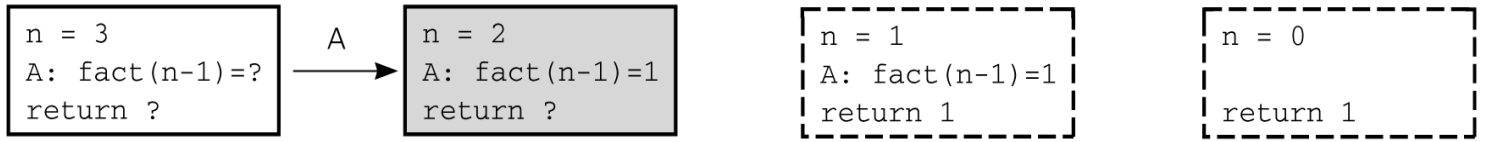


The current invocation of `fact` completes:

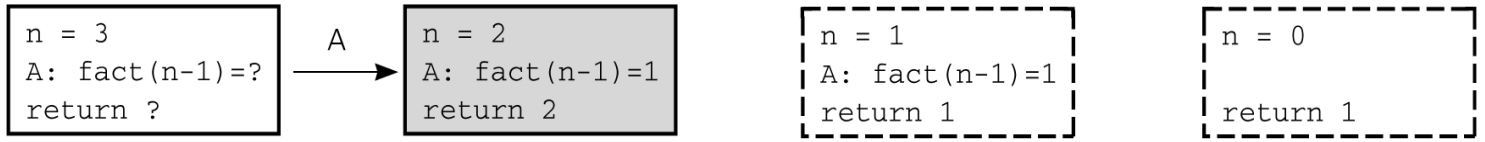


BOX TRACE OF FACT(3)

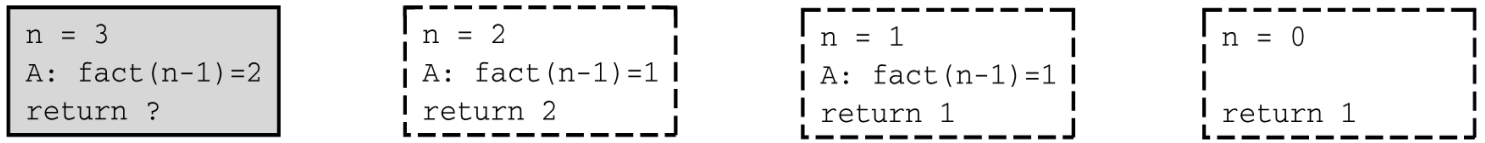
The method value is returned to the calling box, which continues execution:



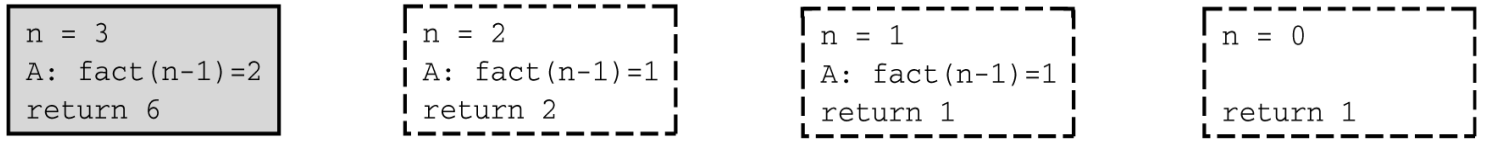
The current invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:



The current invocation `fact` completes:



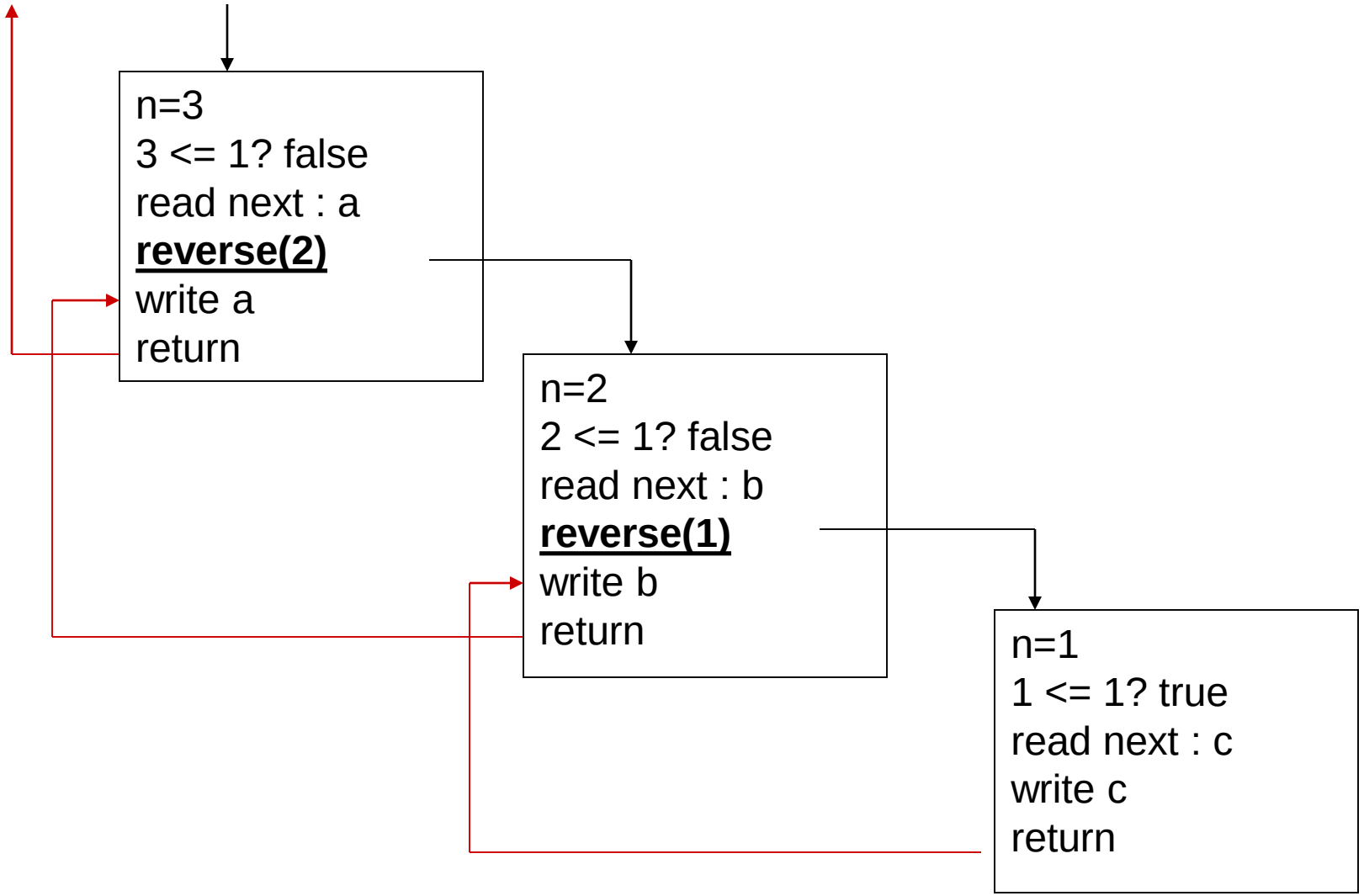
The value 6 is returned to the initial call.

EXAMPLE 2: REVERSE

```
/* reverse(n) reads n characters and prints them in reverse order. */
void reverse(int n)
{
   char next;
   if (n == 1) {                    /* stopping case */
              scanf("%c",&next);
              printf("%c", next);
     }
   else {
              scanf("%c", &next);
              reverse(n-1);
              printf("%c",next);
   }
   return;
}
int main()
{
    printf("Enter a string: ");
    reverse(3);
    printf("\n");
}
```

TRACE OF REVERSE(3)

```
reverse(3);   /* assume input is abc */
```



EXAMPLE 3: FIBONACCI SEQUENCE

- It is the sequence of integers:

t_0		t_1	t_2	t_3	t_4	t_5	t_6	…
0	1	1	2	3	5	8	…	

- Each element in this sequence is the sum of the two preceding elements.

- The specification of the terms in the Fibonacci sequence:

$$t_n = \begin{cases} n & \text{if } n \text{ is } 0 \text{ or } 1 \text{ (i.e. n < 2)} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

EXAMPLE 3: FIBONACCI SEQUENCE

```
int fibonacci(int n)
{
  if (n < 2)
       return n;
  else
       return(fibonacci(n-2) + fibonacci(n-1));
}
```

COMPLEXITY OF FIBONACCI

- This is an example of non-linear recursion. Because total number of recursive calls grows exponentially.

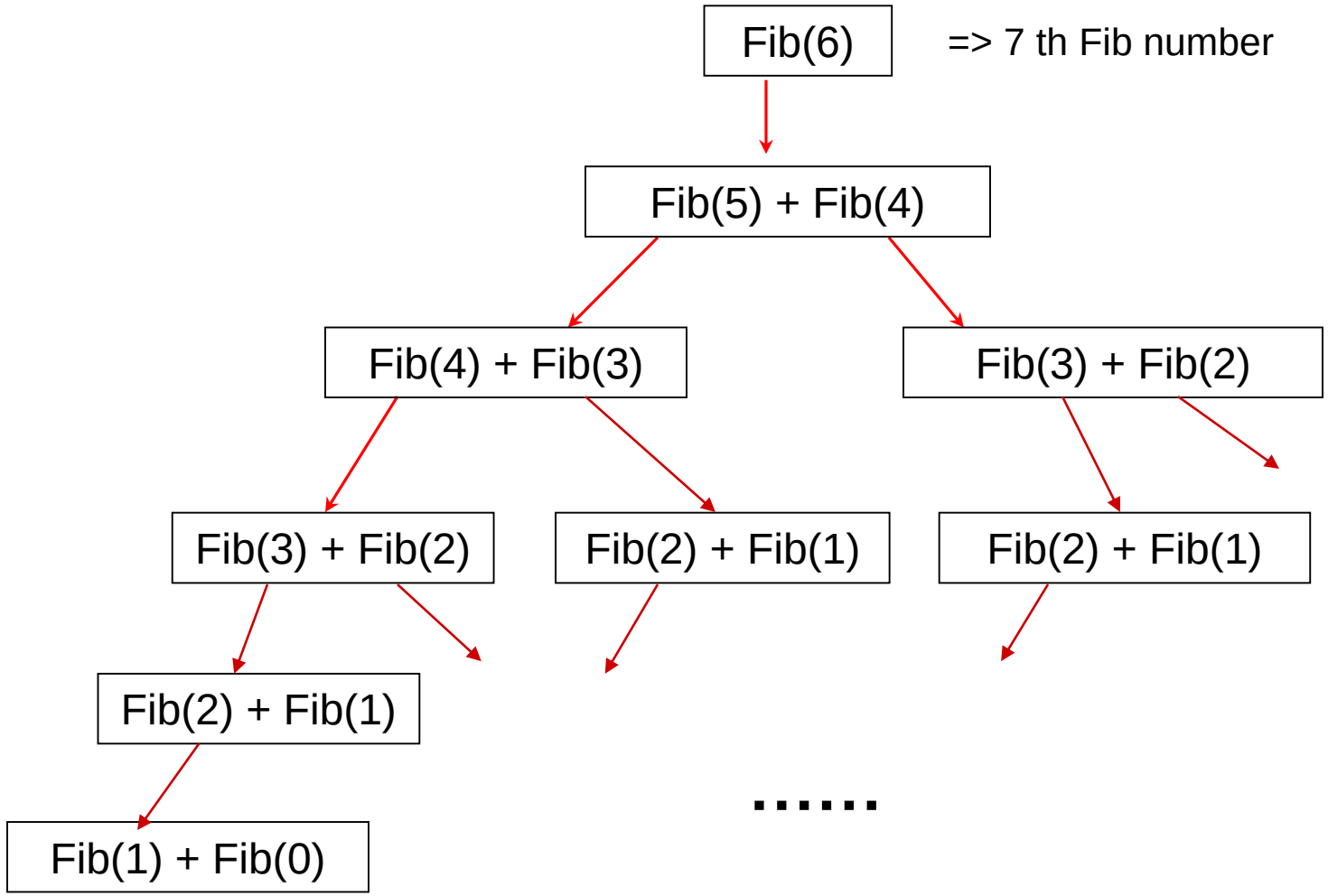
- In the recursive call

```
fib(n) = fib(n-1) + fib(n-2)
```

`fib(n-1)` expression must be evaluated completely before its value can be added to the expression
`fib(n-2)` which must also be evaluated completely.

- Recursion tree is useful in tracing the values of variables during non-linear recursion.

EXAMPLE RECURSION TREE



ITERATIVE VERSION

```
int Fib(int n)
{
    int Prev1, Prev2, Temp, j;

    if (n==0 || n== 1)
        return n;
    else {
        Prev1=0;
        Prev2 = 1;
        for (j=1; j <= n; j++)
        {
            Temp = Prev1 + Prev2;
            Prev2 = Prev1;
            Prev1 = Temp;
        }
        return Prev1;
        }
}
```

COMPARISON OF ITERATION AND RECURSION

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a function is avoided in iterative version.

- However a recursive solution can be sometimes the most natural and logical way of solving a problem.
  - Conflict: machine efficiency versus programmer efficiency.

- It is always true that recursion can be replaced with iteration and a stack. (and vice versa)

EXERCISES: 1

Trace the following recursive function:

```
#include <stdio.h>
int f(int c)
{
    if (c <= 10) {
      printf("%d\n", c);
      f(c + 1);
    }
}

int main(){
    f(0);
}
```

EXERCISES : 2

```
#include <stdio.h>
void f(int);
void g(int);

void f(int c)
{
    printf("hello from f()\n");
    if (++c <= 3)
      g(c);
}
```

```
void g(int c)
{
    printf("hello from g()\n");
    f(c);
}

int main()
{
   printf("hello from main\n");
   f(1);
   return 0;
}
```

EXERCISES : 3

Analyze the output of the following program

```
#include <stdio.h>
int g(int n, int x, int y)
{
  return n == 0 ? x : g(n-1, y, x+y);
}
int f(int n)
{
  return g(n, 0, 1);
}
int main(void){
  int i;
  for (i=1; i<=10; i++)
    printf("%d\n", f(i));
  return 0;
}
```

EXERCISES : 4

Write ***isPalindrome(…)*** recursive function.

BINARY SEARCH- *RECURSIVE*

```
İnt binsearch(int list[], int searchnum, int left, int right)
{
      int middle;
      if(left <= right){
          middle = (left + right)/2;
              switch(compare(list[middle], searchnum)){
                        case -1 : return binsearch(list, searchnum, middle+1, right);
                        case 1: return binsearch(list, searchnum, left, middle-1);
                        case 0;  return middle;
              }
       }
       return -1 ;
}
```

PERMUTATIONS

Given a set of n>=1 elemets, print out all possible permutations of this set.

n!

{a,b,c}‘s permutations

{a, b, c}

{a, c, b}

{b, a, c}

{b, c, a}

{c, a, b}

{c, b, a}

```
void perm(char *list, int i, int n)
{
        int j, temp;
        if(i == n){
                for(j = 0 ; j <=n ; j++)
                        printf("%c", list[j]);
                 printf("\n");
        }
        else{
              for(j = i ; j <= n; j++){
                    SWAP(list[i], list[j], temp);
                     perm(list, i+1, n);
                    SWAP(list[i], list[j], temp);
              }
        }
}
```