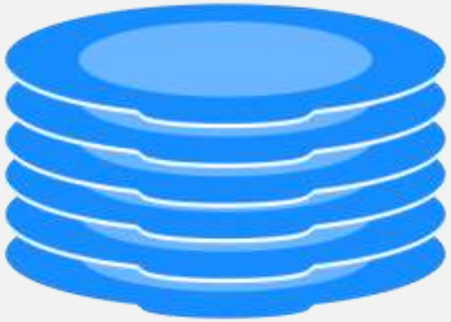
STACK

WHAT IS STACK ??

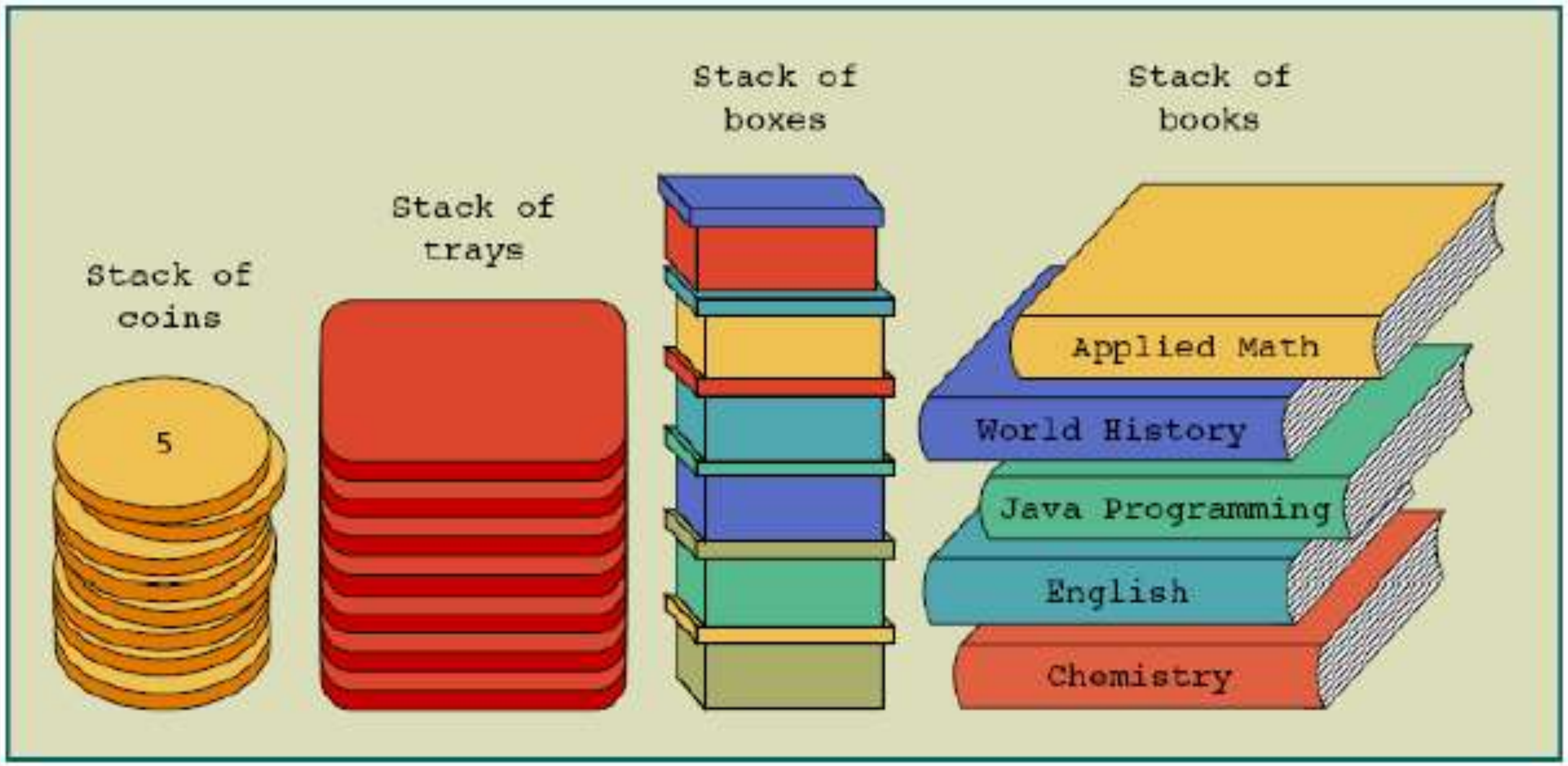
- It is an ordered group of homogeneous items of elements.
- Stacks are dynamic data structures that follow the Last In First Out (LIFO) principle.
- It is a linear list where all insertions and deletions are permitted only at one end of the list.

Analogy :

- It is just like a pile of plates kept on top of each other.
- Think about the things you can do with such a pile of plates
  - Put a new plate on top
  - Remove the top plate
- If you want the plate at the bottom, you have to first remove all the plates on top. Such an arrangement is called **Last In First Out -** the last item that was placed is the first item to go out.

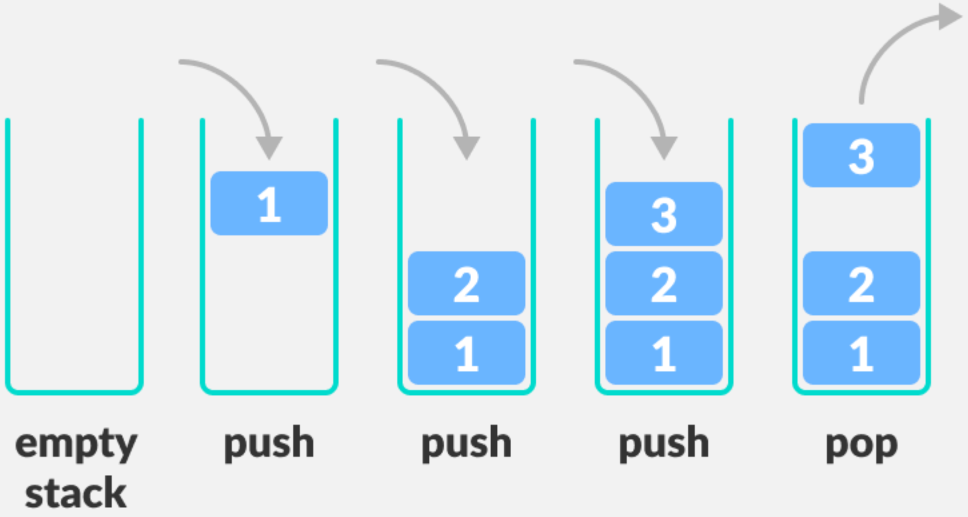


CONCEPTUAL STACKS



LIFO PRINCIPLE OF STACK

- **Push**
  - Push operation is used to add new elements in to the stack.
- **Pop**
  - Pop operation is used to delete elements from the stack.

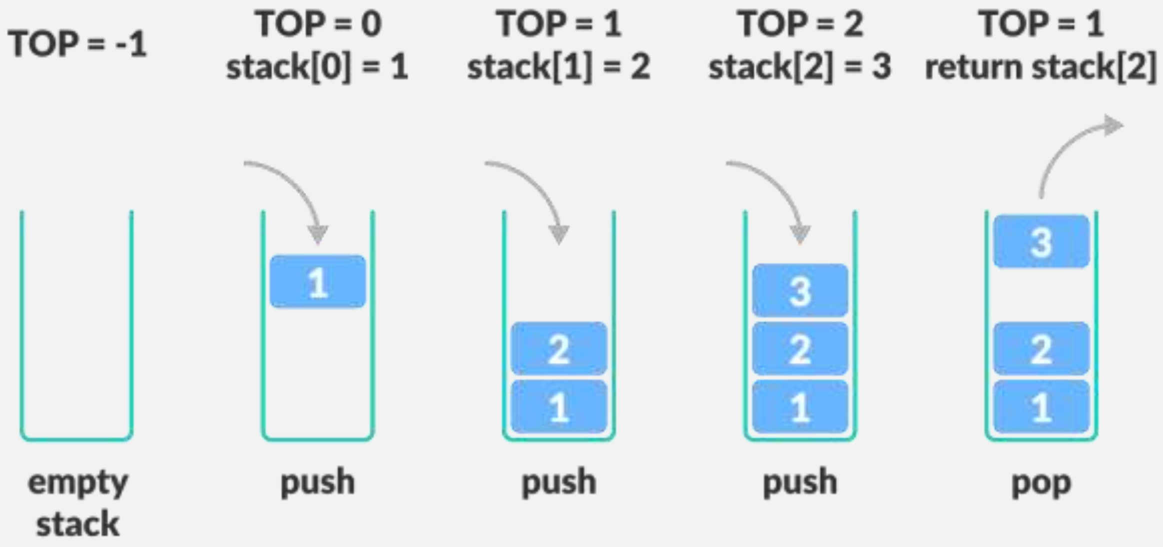


Stack Push and Pop Operations

Note: Both insertion and removal are allowed at only one end of Stack called **Top**

WORKING OF STACK DATA STRUCTURE

- A pointer called TOP is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
- On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
- On popping an element, we return the element pointed to by TOP and reduce its value.
- Before pushing, we check if the stack is already full
- Before popping, we check if the stack is already empty
- When a stack is completely full, it is said to be **Overflow state** and if stack is completely empty, it is said to be **Underflow state**.



OPERATIONS ON STACK-PUSH

Push ( ) : Insert an element

Algorithm

Step 1: Checks if the stack is full.

Step 2: If the stack is full, produces an error and exit.

Step 3: If the stack is not full, increments **top** to point next empty space.

Step 4: Adds data element to the stack location, where top is pointing.

Step 5: Returns success.

```
push()
{
  if(top == MAXSIZE-1)
  {
    Print "Stack Overflow"
  }
  else
  {
     top = top + 1;
     stack[top] = item;
  }
}
```

OPERATIONS ON STACK-POP

Pop( ): Delete an element

Algorithm

Step 1: Checks if the stack is empty.

Step 2: If the stack is empty, produces an error and exit.

Step 3: If the stack is not empty, accesses the data element at which **top** is pointing.

Step 4: Decreases the value of top by 1.

Step 5: Returns success.

```
pop()
{
   if(top == -1)
   {
      Print "Stack is Empty"
   }
   else
   {
       item=stack[top];
       print deleted item
       top = top -1
   }
}
```

OPERATIONS ON STACK-DISPLAY

Display( ): Display all elements

Algorithm

Step1: Checks if the stack is empty.

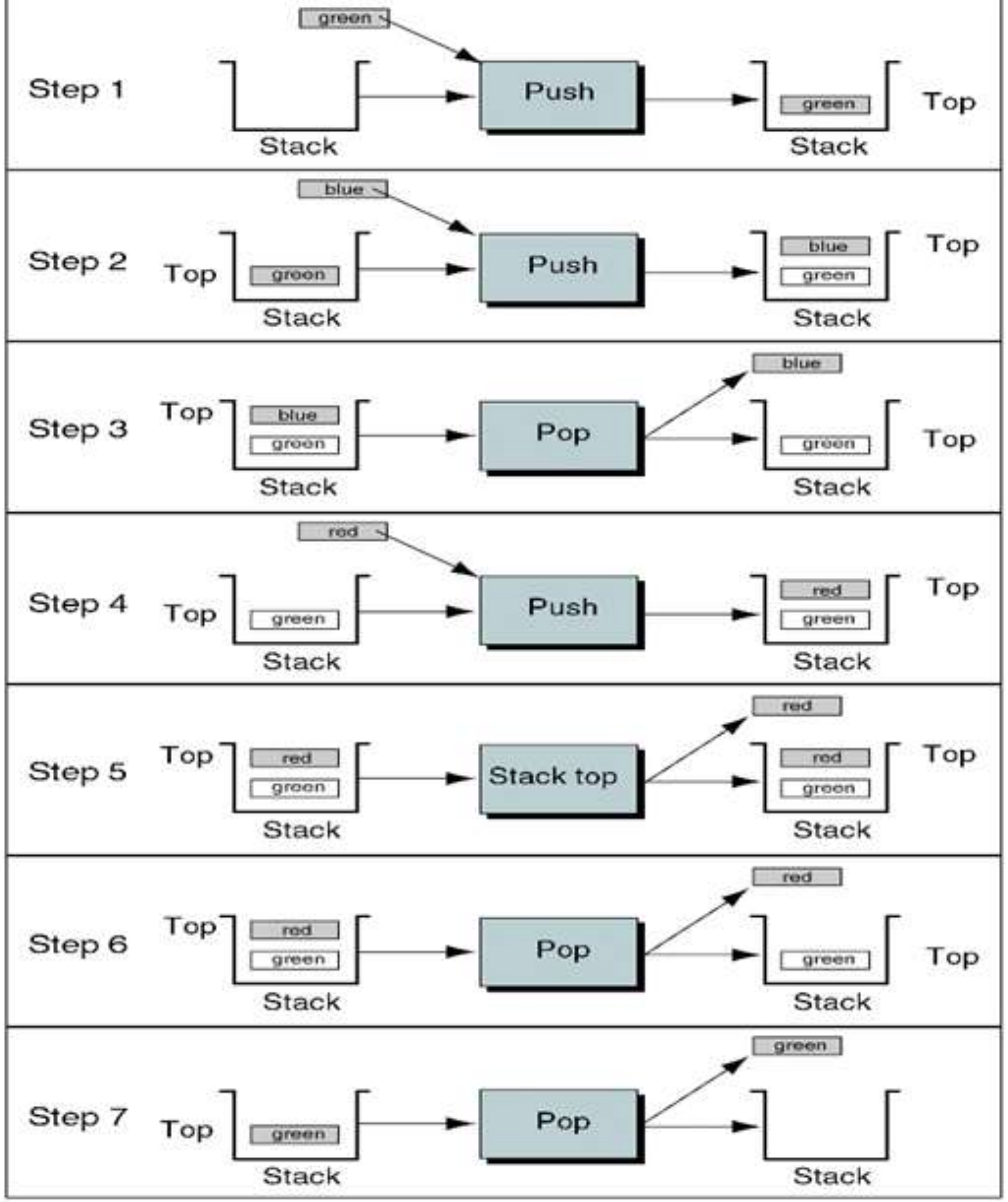
Step2: If the stack is empty, print “Stack Empty” and exit.

Step 3:If the stack is not empty, accesses the data element at which **top** is pointing.

Step 4: Print the elements of the stack.

Step 5: Returns success.

```
display()
{
  if(top == -1)
  {
    Print "Stack is Empty"
  }
  else
  {
    for(i=top;i>=0;i--)
       Print Stack[i]
  }
}
```



STACK EXAMPLE

ARRAY IMPLEMENTATION OF STACK

```c
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
int top=-1;
void push(int a[],int item)
{
    top=top+1;
    a[top]=item;
}
int pop(int a[])
{
    int item;
    item=a[top];
    top=top-1;
    return item;
}
```

```c
void display(int a[])
{
    int i;
    if(top==-1)
        printf("The stack is empty\n");
    else if(top!=-1)
    {
        printf("The stack elements are\n");
        for(i=top; i>=0;i--)
            printf("%d ",a[i]);
        printf("\n");
    }
}
```

```
int main()
{
    int s[10],choice,item;
    while(1)
    {
        printf("Enter the choice\n");
        printf("1 Push\n2 Pop\n3 Display\n4 Exit\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: if(top==SIZE-1)
                    {
                        printf("The stack is full\n");
                        break;
                    }
                    else
                   {
                     printf("Enter the element to be pushed\n");
                     scanf("%d",&item);
                     push(s,item);
                    }
                    break;
case 2: if(top==-1)
                   {
                       printf("The stack is empty\n");
                       break;
                   }
                   item=pop(s);
                   printf("Popped element is %d\n",item);
                   break;

   case 3: display(s);
                       break;
               case 4: exit(0);
           }
       }
       return 0;
   }
```

STRUCTURE IMPLEMENTATION OF STACK

```c
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
struct stack
{
    int top;
    int items[MAX];
};
void push(int,struct stack *);
void pop(struct stack *);
void display(struct stack *);
int main()
{
    struct stack s;
    s.top=-1;
    int choice,item;
    for(;;)
    {
        printf("Enter your choice\n");
        printf("1 Push\n2 Pop\n3 Display\n4 Exit\n");
        scanf("%d",&choice);
```

```
 switch(choice)
         {
              case 1: printf("Enter the
item\n");
                      scanf("%d",&item);
                      push(item,&s);
                      break;
              case 2: pop(&s);
                      break;
              case 3: display(&s);
                      break;
              case 4: exit(0);
         }
    }
    return 0;
}
```

```
void push(int item,struct stack *s)
{
    if(s->top==MAX-1)
        printf("The stack is full\n");
    else
    {
        (s->top)++;
        s->items[s->top]=item;
    }
}
```

```
void pop(struct stack *s)
{
    int item;
    if(s->top==-1)
        printf("The stack is empty\n");
    else
    {
        item=s->items[s->top];
        (s->top)--;
        printf("%d deleted\n",item);
    }
}
```

```
void display(struct stack *s)
{
    int t=s->top;
    if(s->top==-1)
        printf("The stack is empty\n");
    else
    {
        printf("Elements in the stack are\n");
        while(t>-1)
        {
            printf("%d ",s->items[t--]);
        }
        printf("\n");
    }
}
```

APPLICATIONS OF STACK

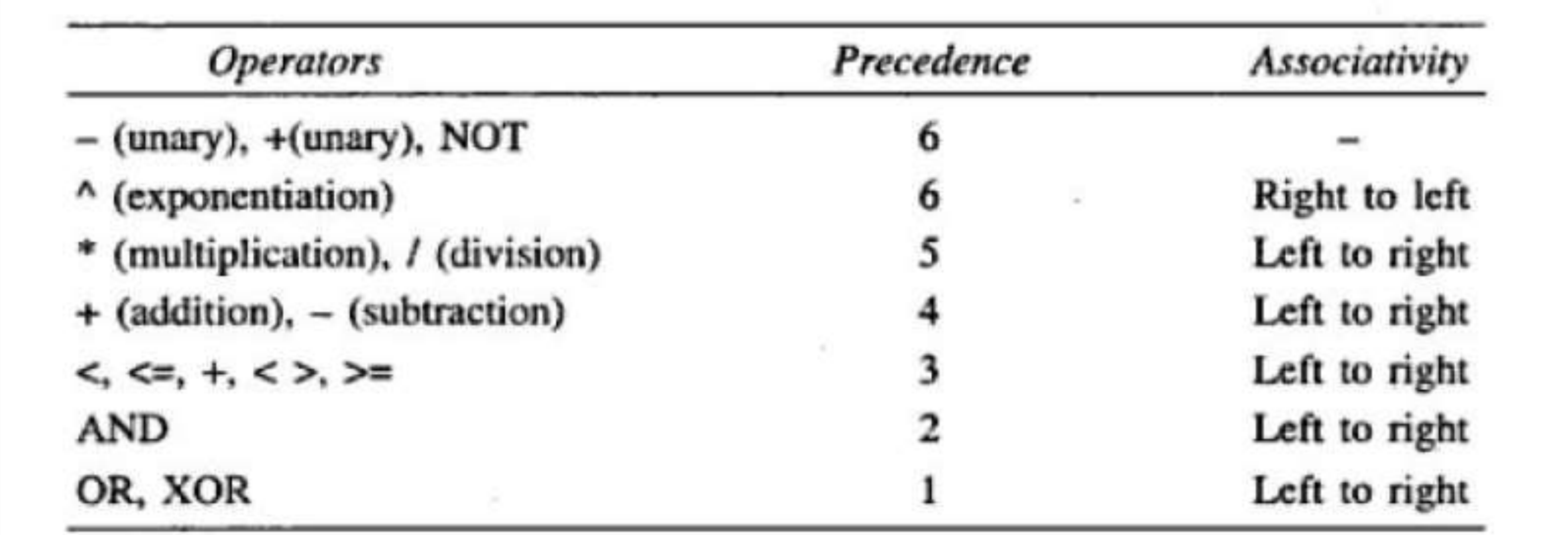
- Conversion of Arithmetic Expression
- Evaluation of Expression
- Recursion

There are three popular methods used for representation of an expression

Infix	A + B	Operator between operands.
Prefix	+ AB	Operator before operands.
Postfix	AB +	Operator after operands.

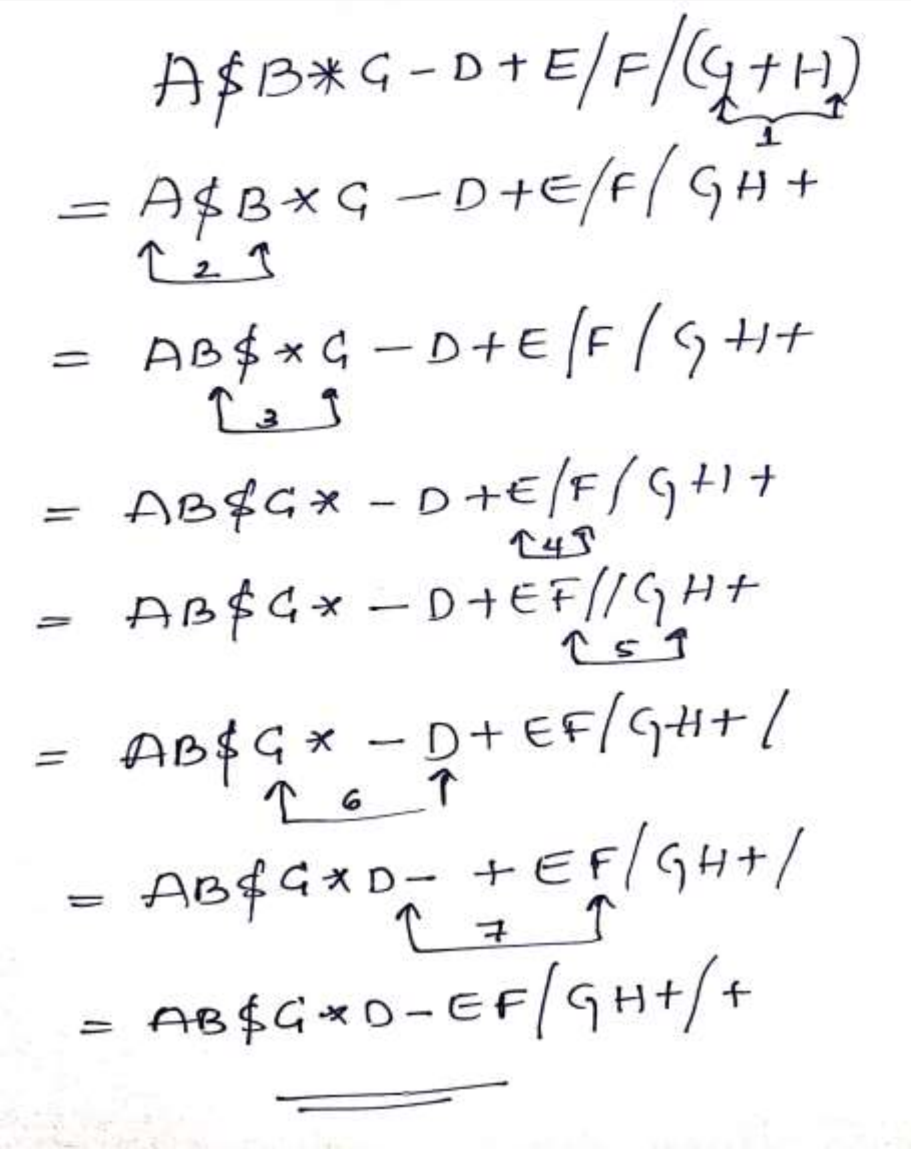
CONVERSION OF EXPRESSION

- An arithmetic expression consists of operands and operators.
- Operands are variables or constants and operators are of various types such as arithmetic unary and binary operators and Boolean operators.
- In addition to these, parentheses such as '(' and ')' are also used.



Operators	*Precedence*	*Associativity*
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right

CONVERSION FROM INFIX TO POSTFIX EXPRESSION



$$A\$B*G-D+E/F/(\underbrace{G+H}_{1})$$

$$=\underbrace{A\$B}_{2}*G-D+E/F/GH+$$

$$=\underbrace{AB\$*G}_{3}-D+E/F/GH+$$

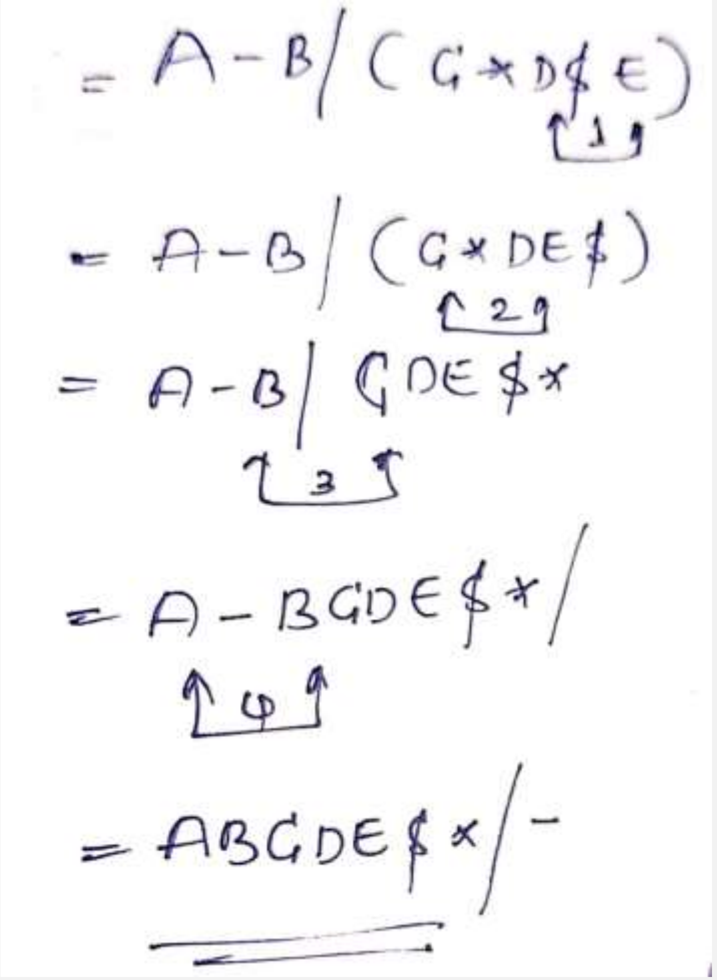
$$=AB\$G*-D+\underbrace{E/F}_{4}/GH+$$

$$=AB\$G*-D+\underbrace{EF//GH+}_{5}$$

$$=AB\$\underbrace{G*-D}_{6}+EF/GH+/$$

$$=AB\$G*\underbrace{D-+EF}_{7}/GH+/$$

$$=AB\$G*D-EF/GH+/+$$



$$=A-B/(G*\underbrace{D\$E}_{1})$$

$$=A-B/(\underbrace{G*DE\$}_{2})$$

$$=A-\underbrace{B/GDE\$*}_{3}$$

$$=\underbrace{A-B}_{4}GDE\$*/$$

$$=ABGDE\$*/-$$

INFIX TO POSTFIX CONVERSION

1. Scan infix expression from left to right.
2. If the scanned symbol is left parentheses “ (“ push in to stack.
3. If the scanned symbol is operand, place it into postfix form.
4. If the scanned symbol is right parentheses “) “ pop all the items from the stack until we get the matching left parentheses and place these items to the postfix form
5. If the scanned symbol is operator, check for the precedence of items at the top of the stack, Symbol at the top of stack is greater than equal to precedence of incoming symbol then pop the item from the stack and place it in postfix form. Push the current symbol onto the stack.

Q (A+(B-C)*D) → ABC-D*+

Symbol	Stack	Postfix
(	(	A
A	(	A
+	(+	A
(	(+(	AB
B	(+(	AB
-	(+(-	ABC
C	(+(-	ABC-
)	(+~~(~~	
*	(+*	ABC-
D	(+*	ABC-D
)		ABC-D*+

2) ((A+(B-C)*D)$ E+F)

Symbol	Stack	Postfix
(	(	
(	((	
A	((	A
+	((+	A
(	((+(	A
B	((+(	AB
-	((+(-	AB
C	((+(-	ABC
)	((+	ABC-
*	((+*	ABC-
D	((+*	ABC-D
)	(	ABC-D*+
$	($	ABC-D*+
E	($	ABC-D*+E
+	(+	ABC-D*+E$
F	(+	ABC-D*+E$F
)		ABC-D*+E$F+

ALGORITHM

```
Algorithm Infix_to_POstfix_Conversion
{
       s=empty stack
       char infix[n]
       char postfix[100]
       while(scan to the end of the string until we reach the last character of infix)
       {
              symbol = next input character
              if(symbol is operand)
                add the symbol to the postfix form
              else if(symbol is '(')
                push the symbol to the stack
              else if(symbol is ')')
              {
                     while((t=pop())!='(')
                     {
                            place the poped t items to the postfix form
                     }
              }
```

```
            else
            {
                  while((the stack is not empty) &&(((precedence of (top of element
of stack))>=(precedence(incoming symbol)))
                  {
                        t=pop()
                        add t to the postfix form
                  }
                  push(symbol)
            }
            while( stack is not empty)
            {
                  t=pop()
                  add poped element t to the postfix string
            }
            print the postfix expression
      }
```

Write a c program to convert and print a given valid parenthesized infix arithmetic expression to postfix expression. the expression consists of single character operands and +,-, *, / operators.
constraints: only four operators used +, -, *, / .

```
#define max 100
#define TRUE 1
#define FALSE 0
struct stack
{
   int top;
  char items[max];
};
struct stack s;
char infix[max],postfix[max];
int pos=0;
void convert();
void push(char);
char pop();
```

```
int precedence(char);
int f=0;
int empty();
int stackfull();
int main()
{
    s.top=-1;
    printf("Enter the infix expression\n");
    gets(infix);
    convert();
    if(f==0)
    {
     printf("The postfix expression is\n");
     puts(postfix);
    }
    return 0;
 }
```

```
void convert()
{
    if(infix[0]==’\0’)
     {
       f=1;
       printf(“Invalid input\n”);
       return;
     }
    int i;
    char symbol,temp;
    for(i=0; infix[i]!='\0';i++)
    {
       symbol=infix[i];
       switch(symbol)
       {
            case '(': push(symbol);
                      break;
            case ')': while((temp=pop())!='(')
                          postfix[pos++]=temp;
                      break;
```

```
            case '+':
            case '-':
            case '*':
            case '/':
            case '$':
while(!empty()&&precedence(s.items[s.top])
>=precedence(symbol)&&precedence(symbol)!=
-1)
                       {
                        temp=pop();

postfix[pos++]=temp;
                      }
                     push(symbol);
                     break;
```

```
   default: if(!isalpha(symbol))
         {
           printf("Invalid input\n");
           f=1;
           return;
          }
          else
          {
            postfix[pos++]=symbol;
            break;
          }
     }
}
```

```
while(!empty())
    {
       temp=pop();
       postfix[pos++]=temp;
    }
}
void push(char ele)
{
    if(stackfull())
        printf("Stack is full\n");
    else
        s.items[++s.top]=ele;
}
```

```
char pop()
{
   if(empty())
   {
     printf("Stack is empty\n");
      exit(0);
    }
   else
       return(s.items[s.top--]);
}
int stackfull() {
    if(s.top==max-1)
        return TRUE;
    else
        return FALSE;
}
```

```
int empty() {
   if(s.top==-1)
      return TRUE;
    else
        return FALSE;
}

int precedence(char symbol)
{
   switch(symbol)
    {
        case '$':return 3;
        case '*':
        case '/':return 2;
        case '+':
        case '-':return 1;
        case '(':
        case ')':return(0);
        default: printf("Invalid input\n");
                 return -1;
    }
}
```

PROCEDURE FOR EVALUATING POSTFIX EXPRESSION

1. Scan the symbol from left to right
2. If the scanned symbol is an operand, push it on the stack
3. If the scanned symbol is an operator, pop two elements from the stack. The first popped element is operand is Operand2 and second popped element is Operand1

   This can be achieved using statements:

   Op2=s[top--]

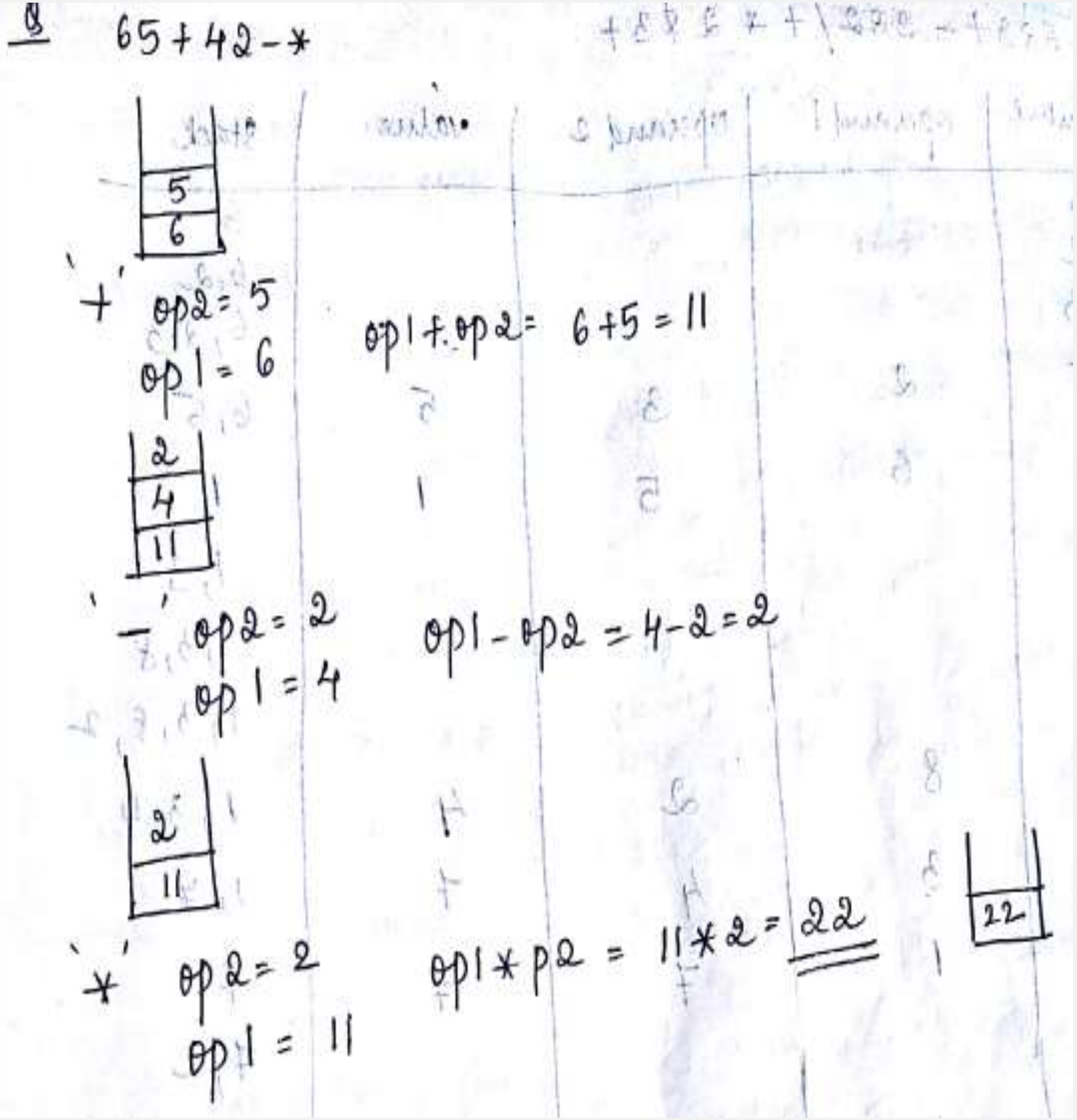
   Op1=s[top--]
4. Perform the indicated operation

   Res = Op1 op Op2
5. Push the result on to the stack
6. Repeat the above procedure till the end of input is encountered.

ALGORITHM

```
S = emptystack
while(input string is not reached to the end)
{
        symbol = next input character
        if(symbol == operand)
        {
                push(&s,symbol)
        }
        else
        {
                op2=pop(&S)
                op1=pop(&S)
                value =  perform the arithmetic operation
                push (&s,value);
        }
}
return (pop(&S));
end of algorithm
```

EXAMPLE



Q 65+42-*

5
6

'+' op2 = 5 op1+op2 = 6+5 = 11

op1 = 6

2
4
11

'-' op2 = 2 op1-op2 = 4-2 = 2

op1 = 4

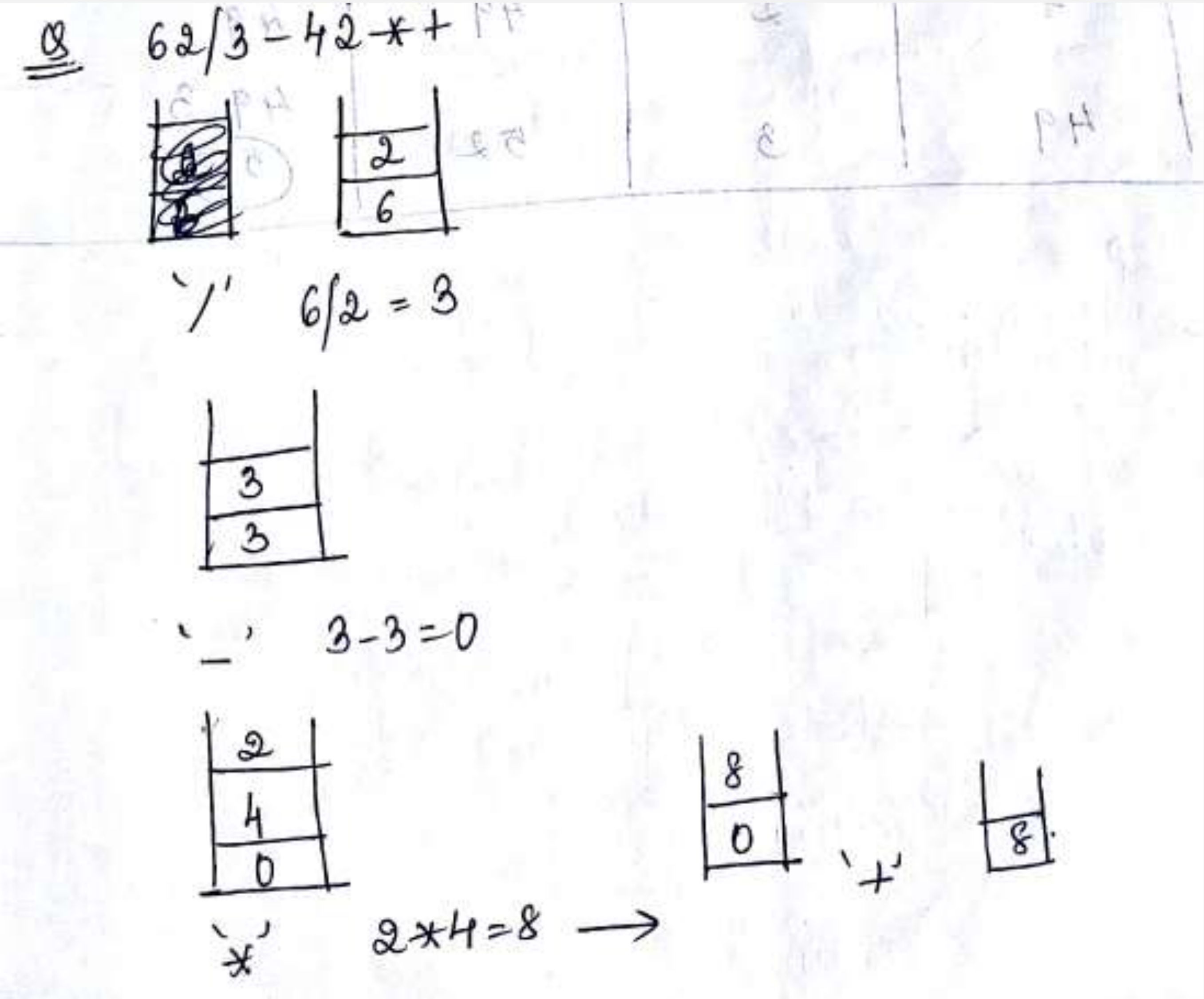
2
11

'*' op2 = 2 op1*p2 = 11*2 = $\underline{\underline{22}}$

op1 = 11

22

EXAMPLE



Q 62/3-42-*+

'/' 6/2 = 3

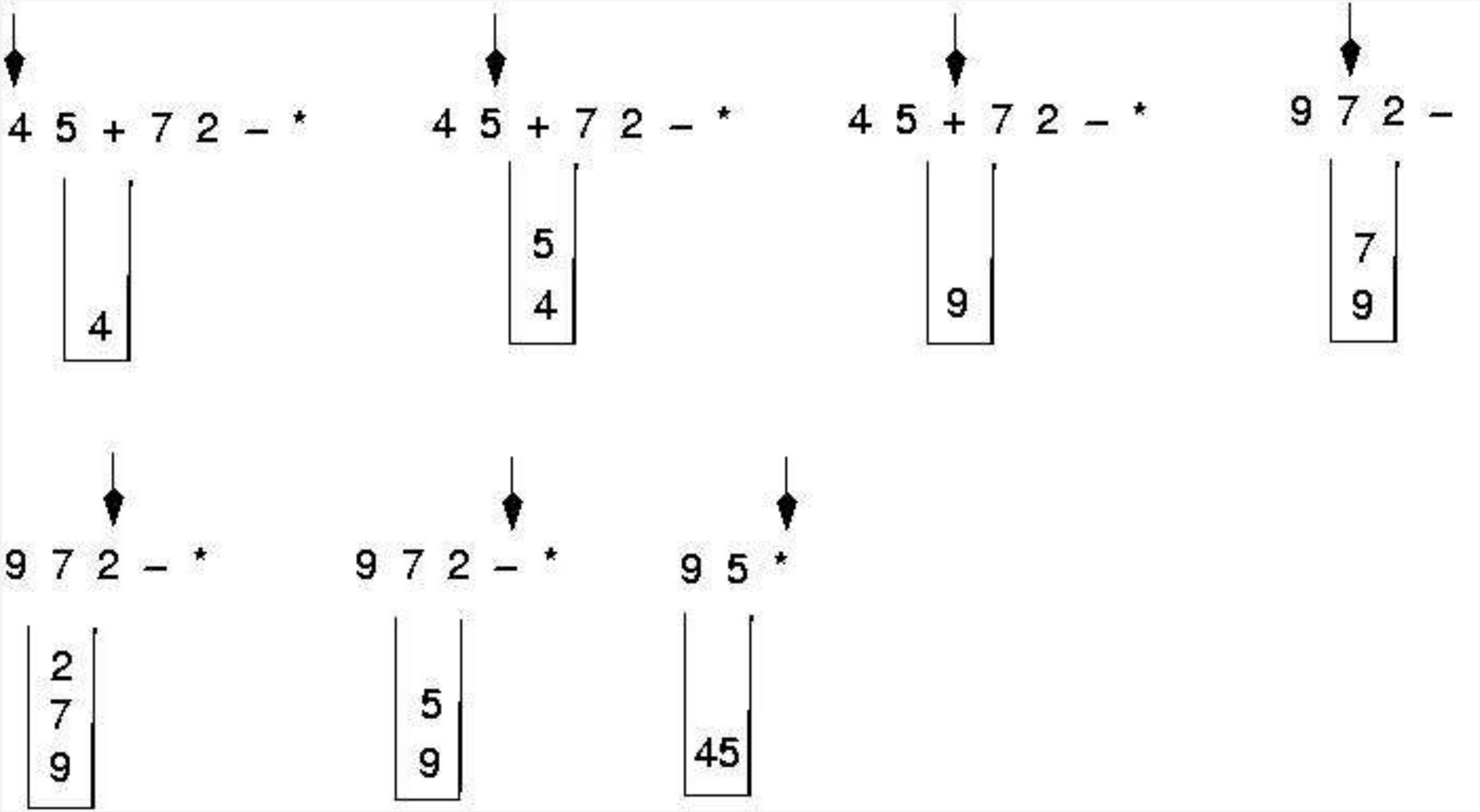
'-' 3-3=0

'*' 2*4=8 →

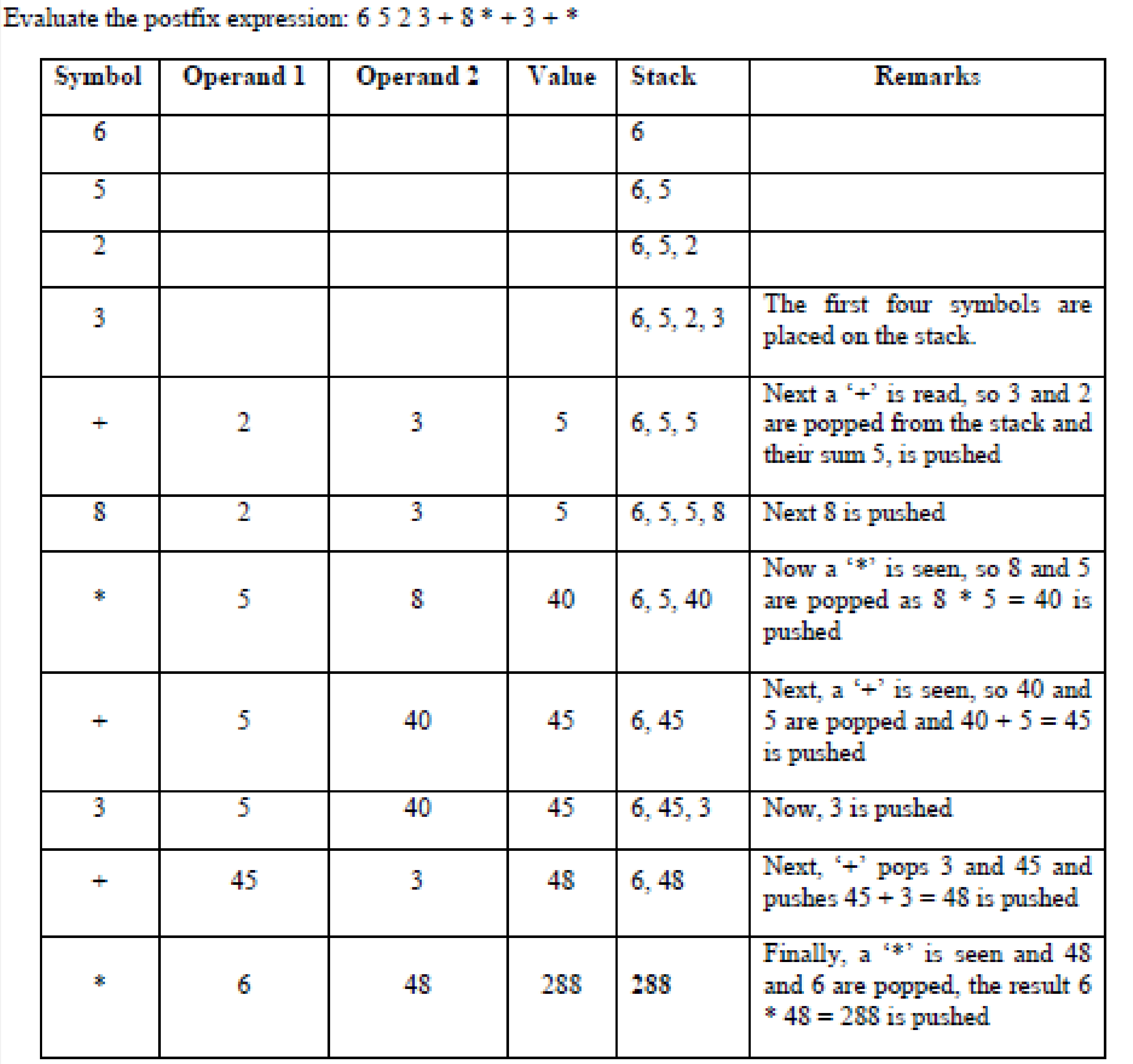
'+'

2
6
3
3
2
4
0
8
0
8

EXAMPLE



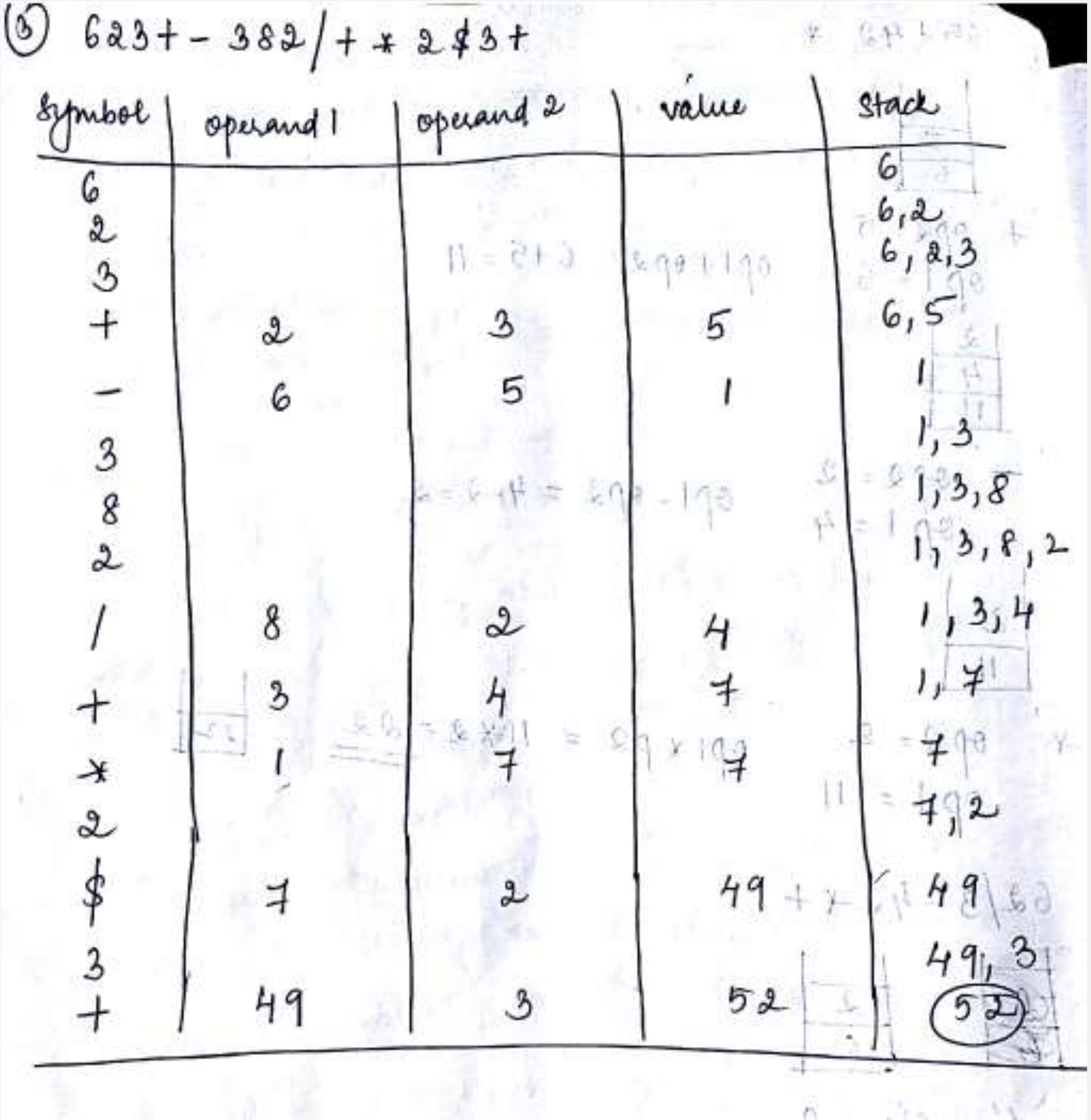
EXAMPLE



Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a ‘+’ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a ‘*’ is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a ‘+’ is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, ‘+’ pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a ‘*’ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

EXAMPLE



(5) 623+-382/+*2$3+

symbol	operand 1	operand 2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3				1,3
8				1,3,8
2				1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2				7,2
$	7	2	49	49
3				49,3
+	49	3	52	52

EVALUATING POSTFIX EXPRESSION

```
  #include<stdio.h>
  #include<stdlib.h>
  #include<ctype.h>
  #include<math.h>
  #include<string.h>
  #define MAX 100
  struct stack
  {
      int top;
      double a[MAX];
  };
  void push(char,struct stack *);
  double pop(struct stack *);
  double compute(double,double,char);

int main()
{
    struct stack s;
    s.top = -1;
    int i;
    double op1,op2,res;
    char postfix[100],symb;
    printf("Enter postfix expression");
    gets(postfix);
```

```
    for(i=0;postfix[i]!='\0';i++)
    {
        symb =postfix[i];
        if(isdigit(symb))
            push(symb,&s);
        else
        {
            op2=pop(&s);
            op1=pop(&s);
            res = compute(op1,op2,symb);
            s.a[++s.top]=res;
        }
    }
    res=pop(&s);
    prinf("Result after evaluation =%f",res);
    return 0;
}
```

```
 void push(char symb,struct stack *s)
 {
     if(s->top == MAX-1)
         printf("Stack Overflow");
     else
         s->a[++s->top]=symp-'0';
 }
 double pop(struct stack *s)
 {
     if(s->top ==-1)
         printf("Stack is Empty\n");
     else
         printf(s->a[s->top--]);
 }

double compute(double op1,double op2,char symb)
{
    switch(symb)
    {
        case '+' : return(op1+op2);
        case '-' : return(op1-op2);
        case '*' : return(op1*op2);
        case '/' : return(op1/op2);
        case '$' :
        case '^' : return(pow(op1,op2));
    }
}
```

RECURSION

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
- A problem-solving approach, that can generate simple solutions to certain kinds of problems that would be difficult to solve in other ways
- Recursion splits a problem into one or more simpler versions of itself
- Using recursive algorithm, certain problems can be solved quite easily.
- Recursive thinking: the general approach

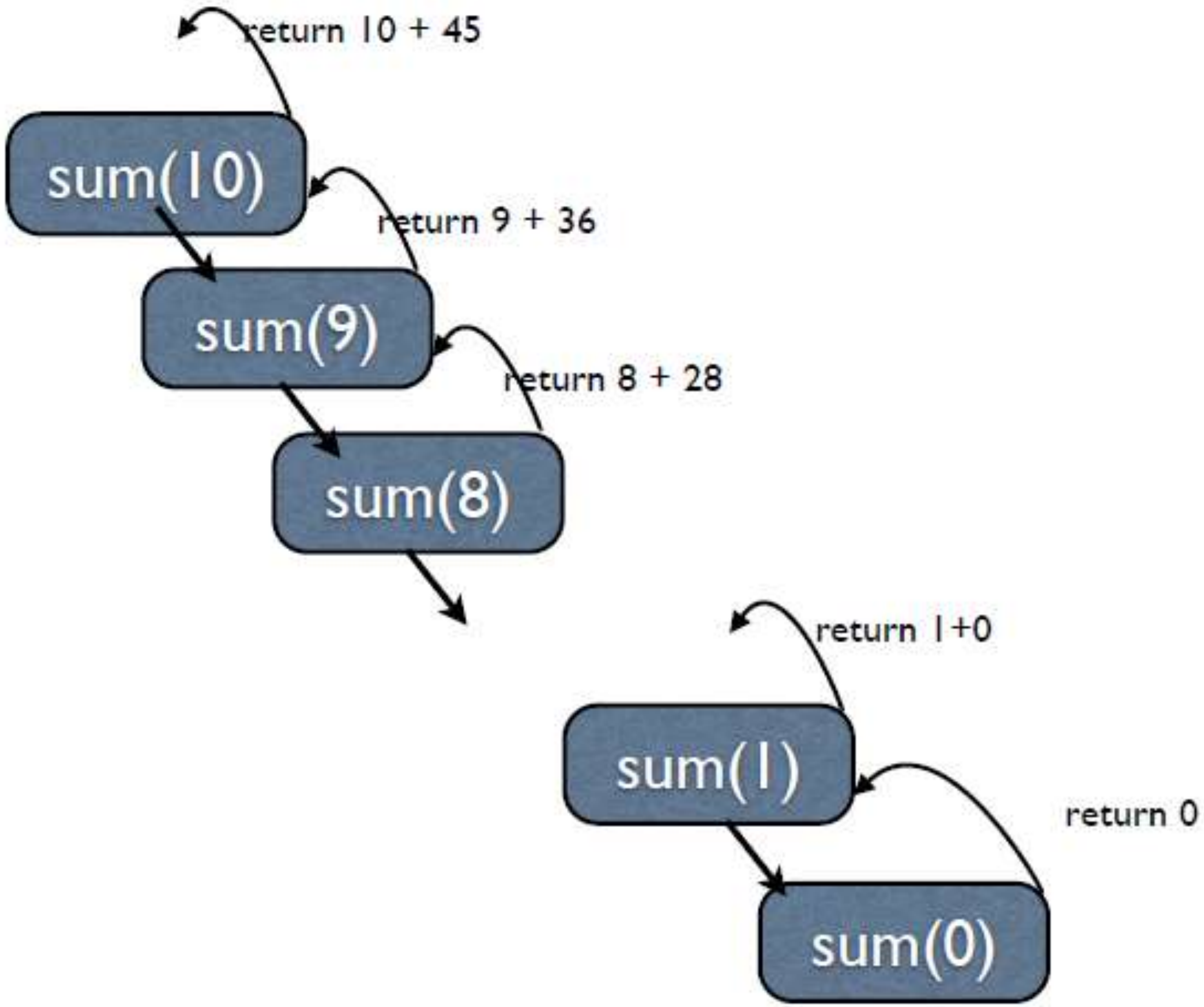
```
if problem is “small enough”
    solve it directly
else
    break into one or more smaller subproblems
    solve each subproblem recursively
    combine results into solution to whole problem
```

EXAMPLE

- Write a function that computes the sum of numbers from 1 to n

```
//with a loop
int sum (int n)
{
 int s = 0;
 for (int i=0; i<n; i++)
    s+= i;
 return s;
}
```

```
//recursively
int sum (int n)
{
  int s;
  if (n == 0)
     return 0;
  else
     s = n + sum(n-1);
  return s;
}
```


return 10 + 45
sum(10)
return 9 + 36
sum(9)
return 8 + 28
sum(8)
return 1+0
sum(1)
return 0
sum(0)

FACTORIAL

- The factorial of a positive number n is given by: factorial of n (n!) = 1 * 2 * 3 * 4 *... * n
- The factorial of a negative number doesn't exist. And the factorial of 0 is 1.

```
//Using iteration
int main()
{
  int i,fact = 1;
  printf("Enter the number:");
  scanf("%d",&num);
  for(i=1;i<=num;i++)
       fact = fact * i;
  printf("Factorial of number = %d",t);
  return 0;
}
```

Example:

num = 4

1st iteration : i=1, fact=1

fact = 1 * 1 = 1

2nd iteration i=2, fact =1

fact = 1 * 2 = 2

3rd iteration : i=3, fact=2

fact = 2 * 3 = 6

4th iteration: i=4, fact=6

fact = 6 *4 = 24

```
//Using Recursion
#include<stdio.h>
int fact(int n);
int main()
{
      int num, t;
      printf("Enter the number:");
      scanf("%d",&num);
      t=fact(num);
      printf("Factorial of number = %d",t);
      return 0;
}
int fact(int n)
{
      if(n==0)
        return(1);
      else
         return(n*fact(n-1));
}
```

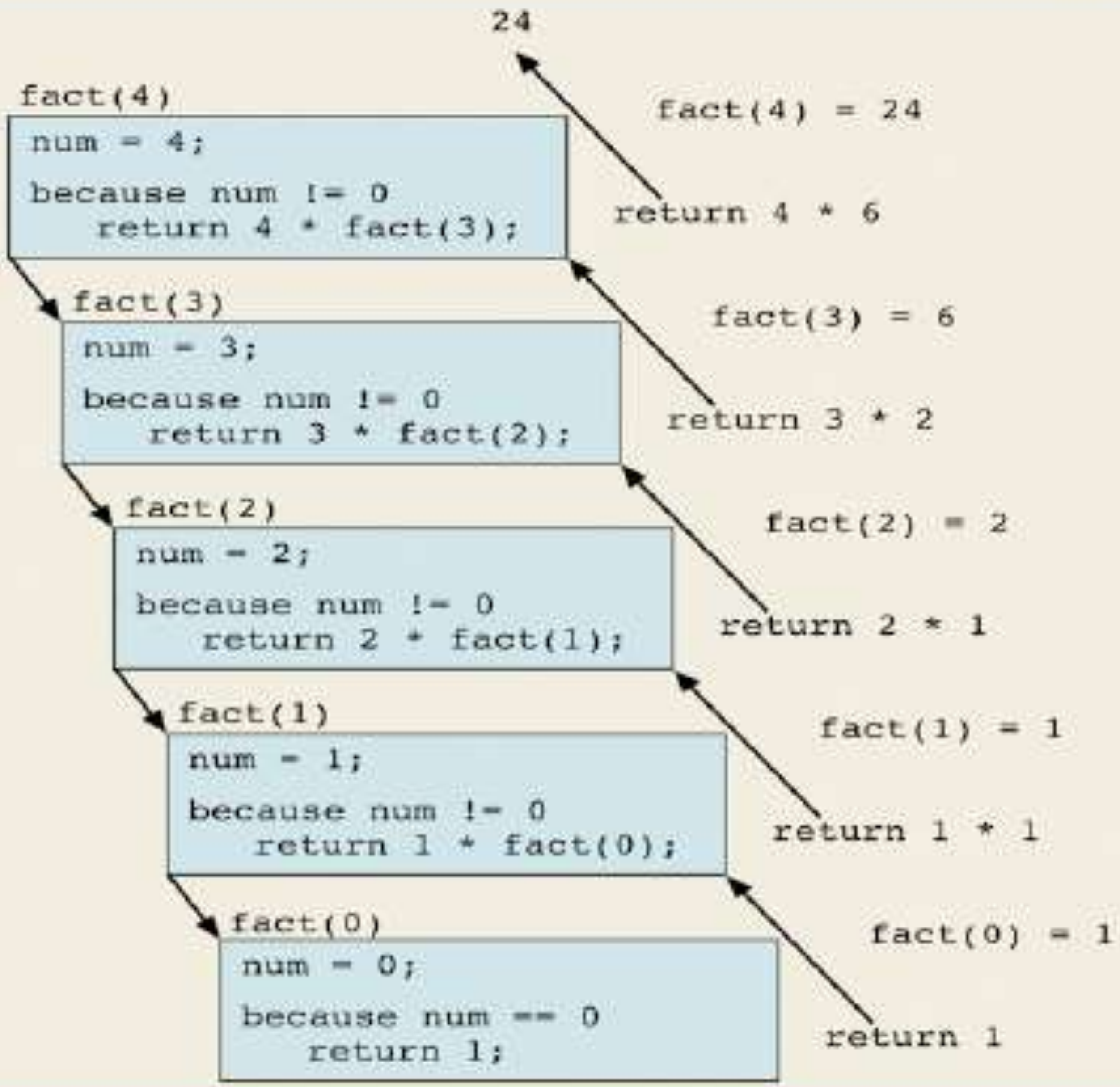
Example:

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1



24

fact(4)

```
num = 4;
because num != 0
   return 4 * fact(3);
```

fact(3)

```
num = 3;
because num != 0
   return 3 * fact(2);
```

fact(2)

```
num = 2;
because num != 0
   return 2 * fact(1);
```

fact(1)

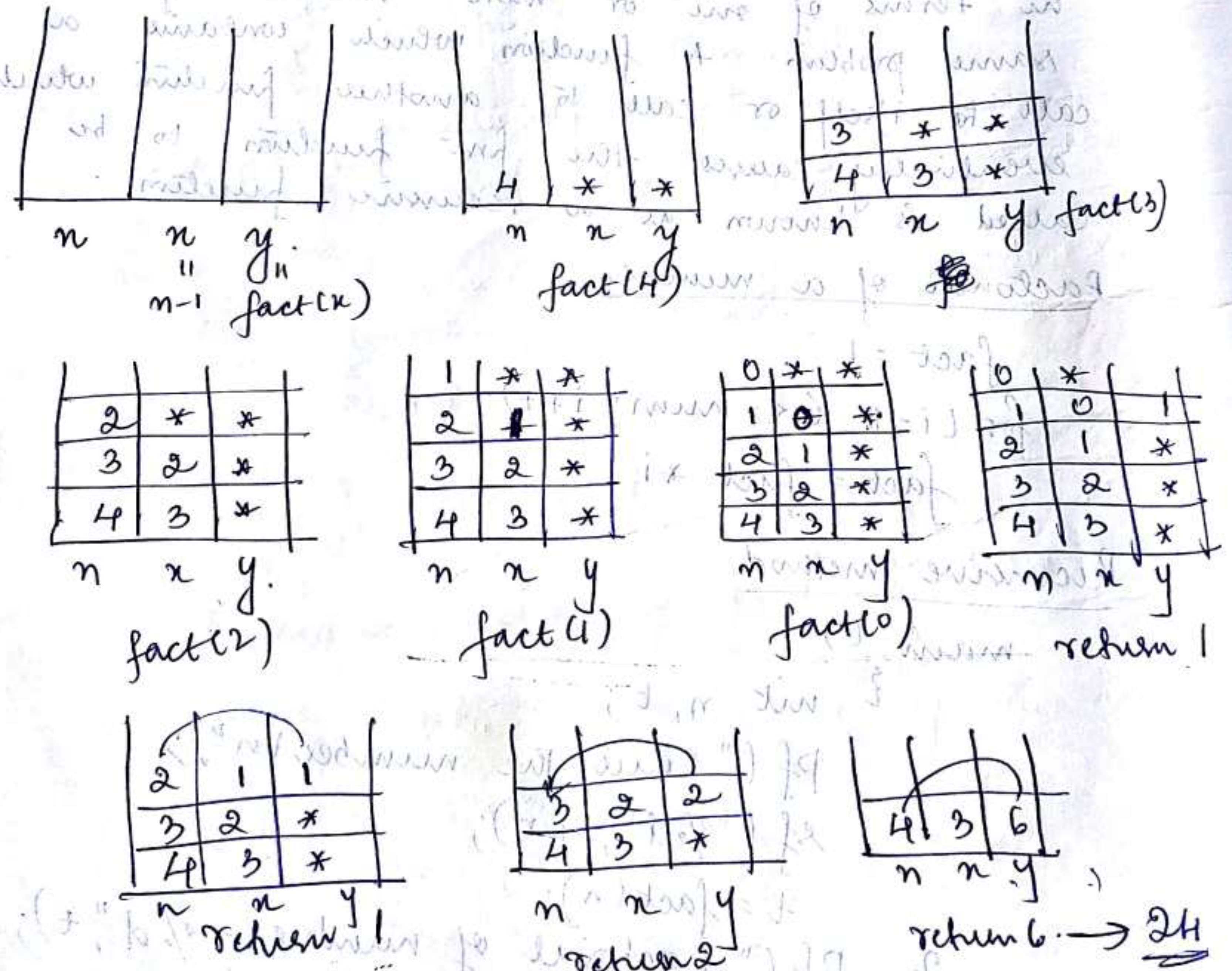
```
num = 1;
because num != 0
   return 1 * fact(0);
```

fact(0)

```
num = 0;
because num == 0
   return 1;
```

```
fact(4) = 24
return 4 * 6
fact(3) = 6
return 3 * 2
fact(2) = 2
return 2 * 1
fact(1) = 1
return 1 * 1
fact(0) = 1
return 1
```

Stack :-



n	x	y

x = n-1, y = fact(x)

n	x	y
4	*	*

fact(4)

n	x	y
3	*	*
4	3	*

fact(3)

n	x	y
2	*	*
3	2	*
4	3	*

fact(2)

n	x	y
1	*	*
2	1	*
3	2	*
4	3	*

fact(1)

n	x	y
0	*	*
1	0	*
2	1	*
3	2	*
4	3	*

fact(0)

n	x	y
0	*	
1	0	1
2	1	*
3	2	*
4	3	*

return 1

n	x	y
2	1	1
3	2	*
4	3	*

return 1

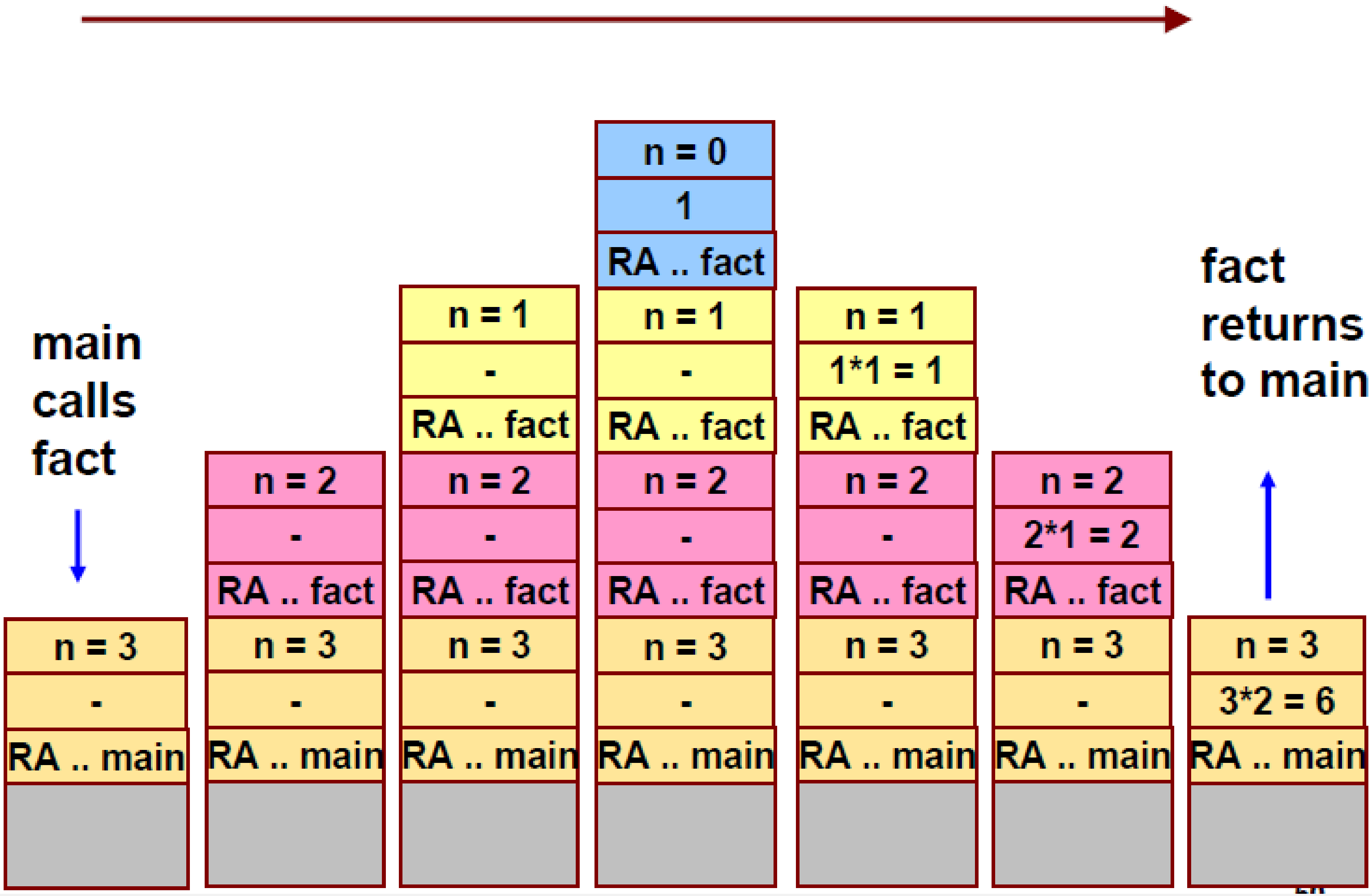
n	x	y
3	2	2
4	3	*

return 2

n	x	y
4	3	6

return 6 → 24

TRACE OF THE STACK DURING EXECUTION



main calls fact

n = 3
-
RA .. main

n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 1
-
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 0
1
RA .. fact
n = 1
-
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 1
1*1 = 1
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 2
2*1 = 2
RA .. fact
n = 3
-
RA .. main

n = 3
3*2 = 6
RA .. main

fact returns to main

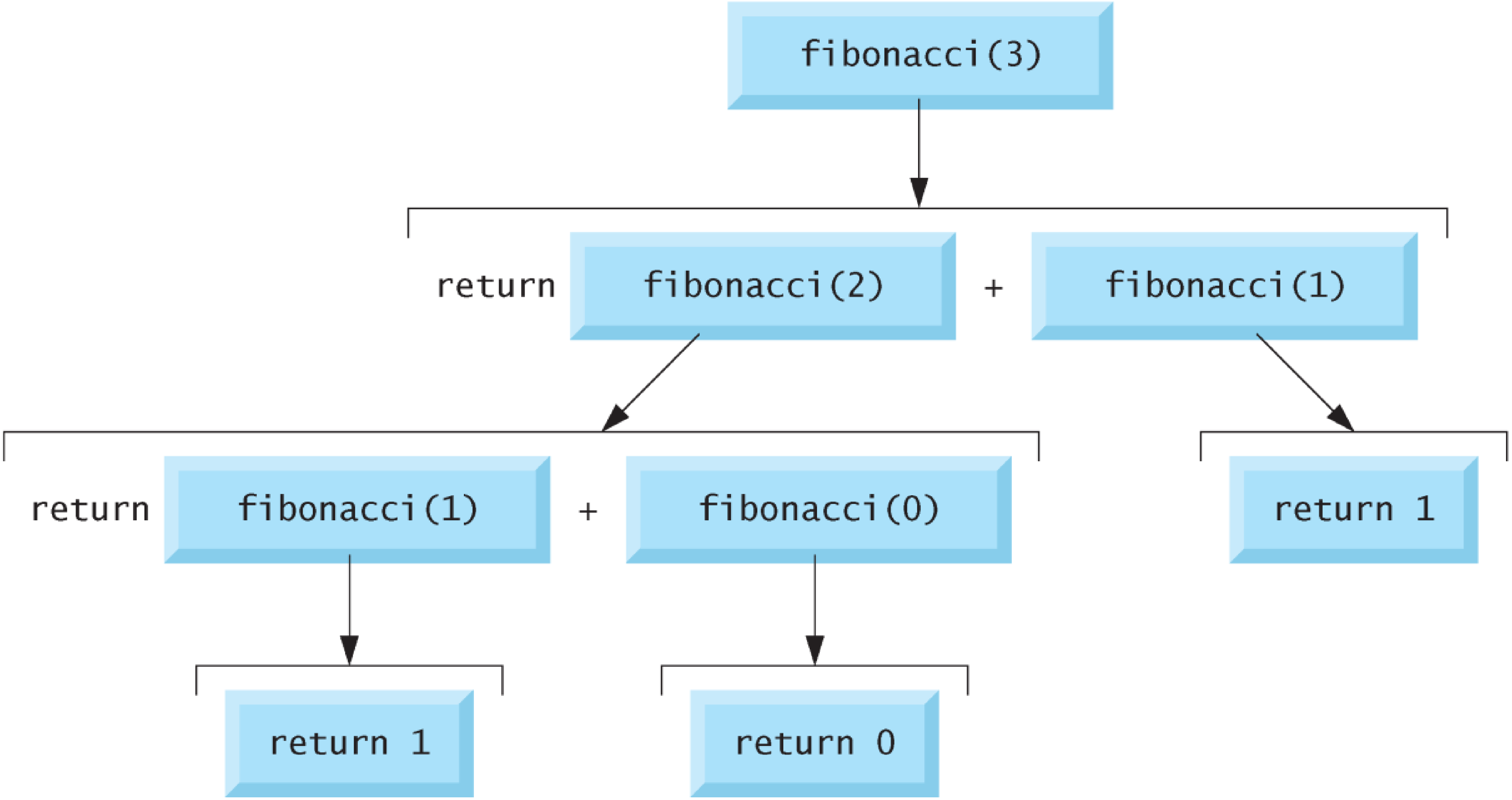
FIBONACCI SERIES BY RECURSION

- The Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- The Fibonacci series may be defined recursively as follows:

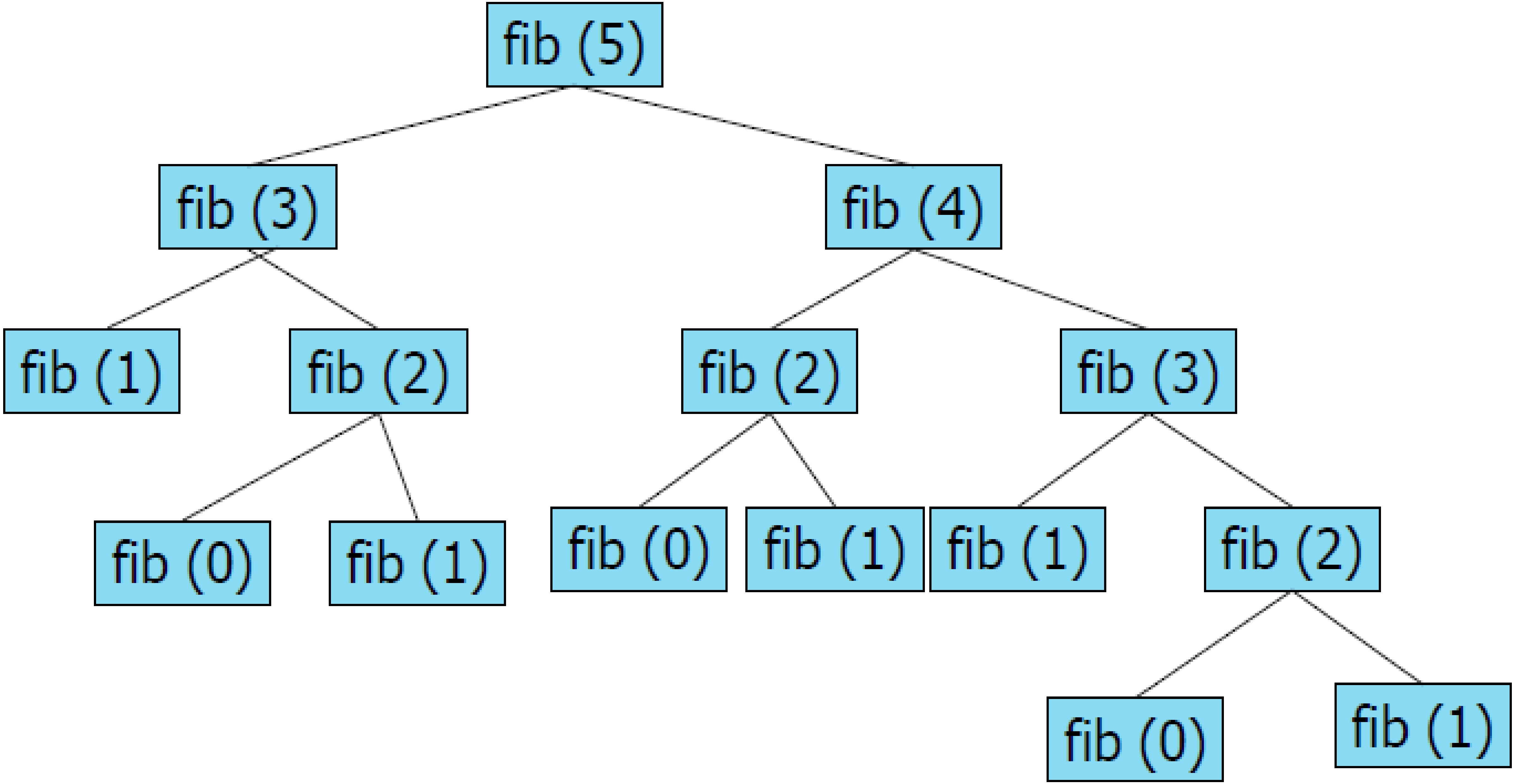
  fibonacci (0) = 0

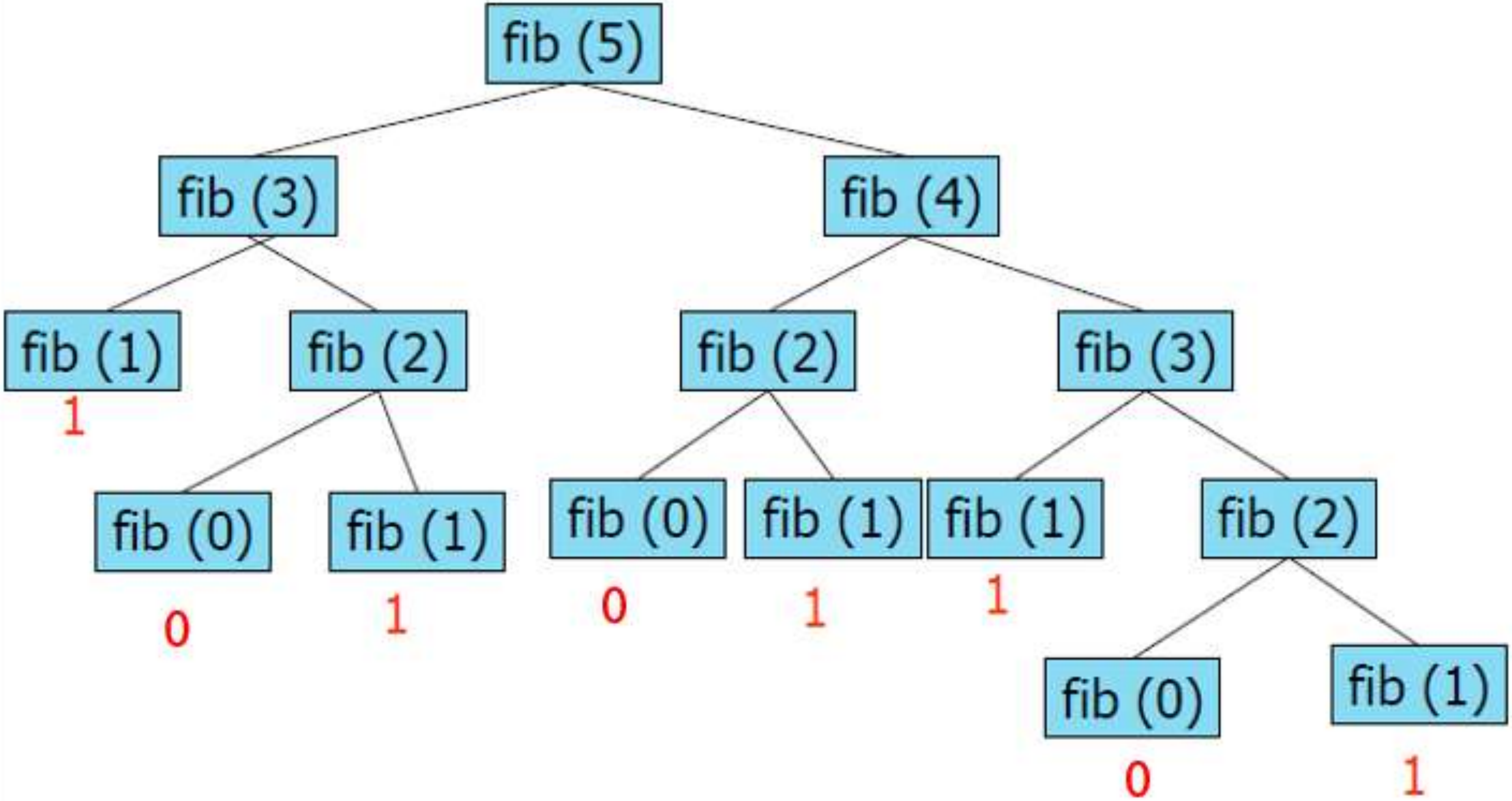
  fibonacci (1) = 1

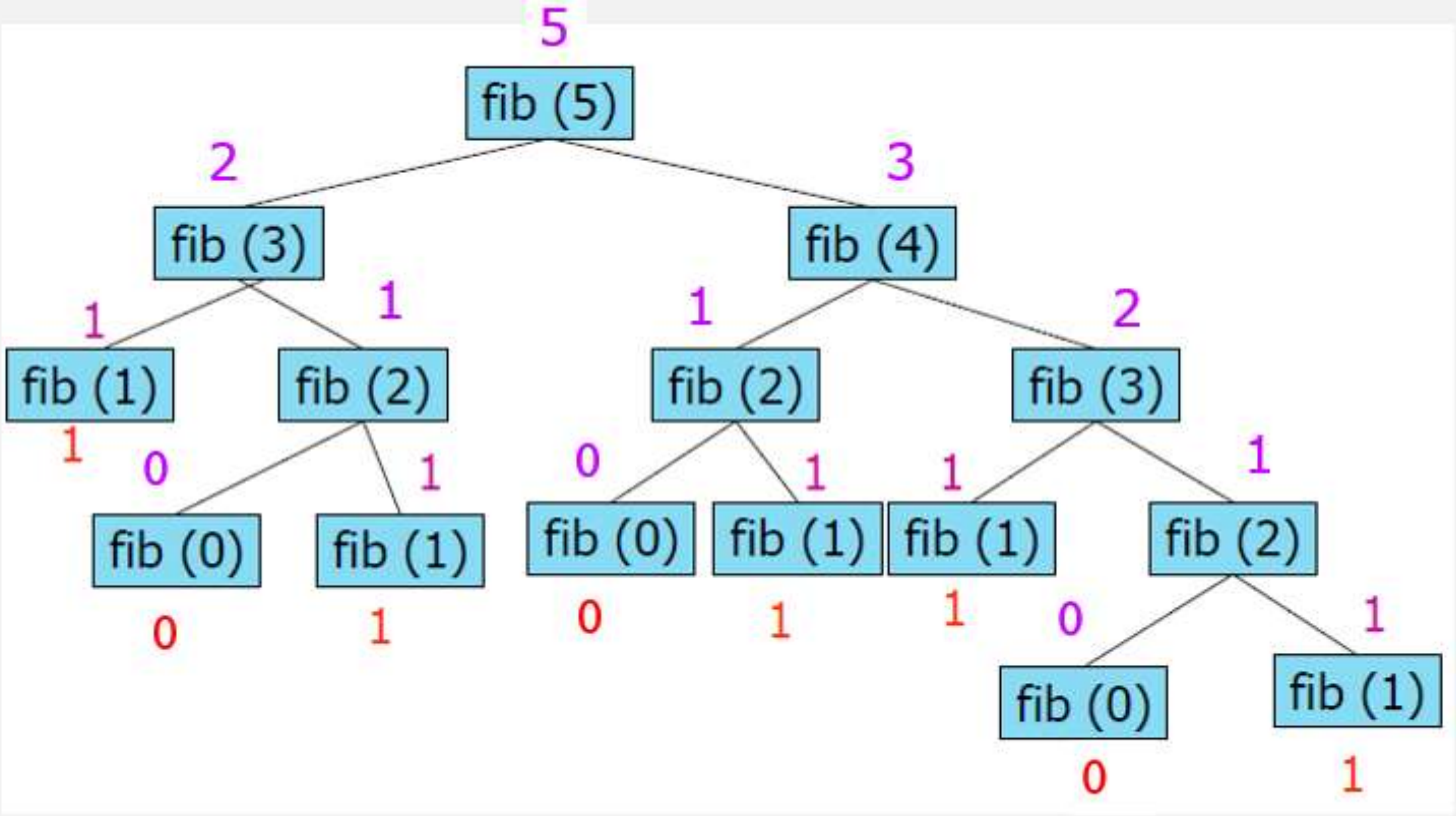
  fibonacci (n) = fibonacci (n – 1) + fibonacci (n – 2)


fibonacci(3)
return
fibonacci(2)
+
fibonacci(1)
return
fibonacci(1)
+
fibonacci(0)
return 1
return 1
return 0

```c
#include<stdio.h>
int fibonacci(int n);
int main()
{
        int n,t;
        printf("Enter a number");
        scanf("%d",&n);
        t=fibonacci(n);
        printf("The %d fibonacci number = %d",n,t);
        return 0;
}
int fibonacci(int n)
{
        if(n==0)
                return 0;
        else if(n==1)
                return 1;
        else
                return (fibonacci(n-1)+fibonacci(n-2))
}
```


fib (5)
fib (3)
fib (4)
fib (1)
fib (2)
fib (2)
fib (3)
fib (0)
fib (1)
fib (0)
fib (1)
fib (1)
fib (2)
fib (0)
fib (1)


fib (5)
fib (3)
fib (4)
fib (1)
fib (2)
fib (2)
fib (3)
1
fib (0)
fib (1)
fib (0)
fib (1)
fib (1)
fib (2)
0
1
0
1
1
fib (0)
fib (1)
0
1



5
fib (5)
2
3
fib (3)
fib (4)
1
1
1
2
fib (1)
fib (2)
fib (2)
fib (3)
1
0
1
0
1
1
1
fib (0)
fib (1)
fib (0)
fib (1)
fib (1)
fib (2)
0
1
0
1
1
0
1
fib (0)
fib (1)
0
1

BINARY SEARCH USING RECURSION

```
#include<stdio.h>
int BinarySearch(int a[],int key,int low, int high)
int main()
{
        int n,i,a[20],key,pos;
        printf("Enter the number of elements:");
        scanf("%d",&n);
        printf("Enter the elements of the array\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter the elements to be searched");
        scanf("%d",&key)
        pos=BinarySearch(a,key,0,n-1);
        if(pos==-1)
                printf(Item/Key not found");
        else
                printf("Item/Key found at %d location",pos+1);
        return 0;
}
```

BINARY SEARCH USING RECURSION

```
int BinarySearch(int a[],int key,int low, int high)
{
        int mid;
        if(low>high)
                return -1;
        mid=(low+high)/2;
        if(key==a[mid])
                return mid;
        if(key<a[mid])
                return(BinarySearch(a,key,low,mid-1));
        else
                return(BinarySearch(a,key,mid+1,high));
}
```

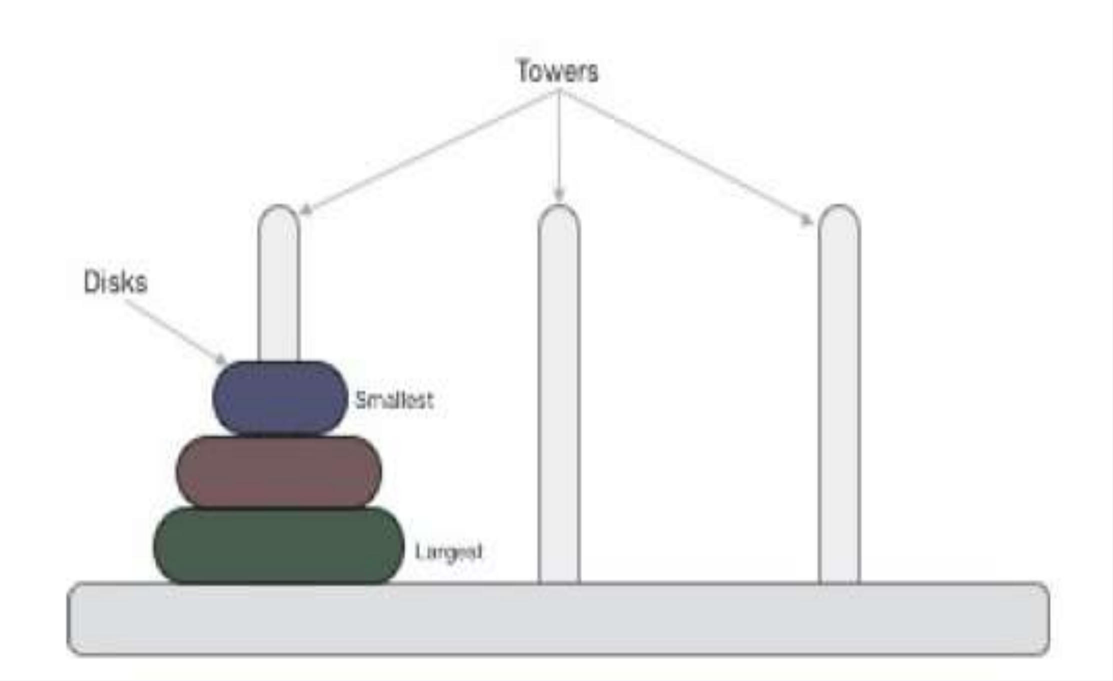
GCD AND LCM USING RECURSION

```
#include<stdio.h>
int gcd(int m, int n);
int main()
{
        int m,n,result,lcm;
        printf("Enter the value of m and n");
        scanf("%d%d",&m,&n);
        result=gcd(m,n);
        printf("GCD of %d and %d = %d",m,n,result);
        lcm=(m*n)/result;
        printf("LCM of %d and %d = %d",m,n,lcm);
        return 0;
}
```

```
int gcd(int m, int n)
{
        int rem;
        if(n==0)
                return m;
        else
        {
                rem=m%n;
                m=n;
                n=rem;
                return(gcd(m,n))
        }
}
```

TOWERS OF HANOI

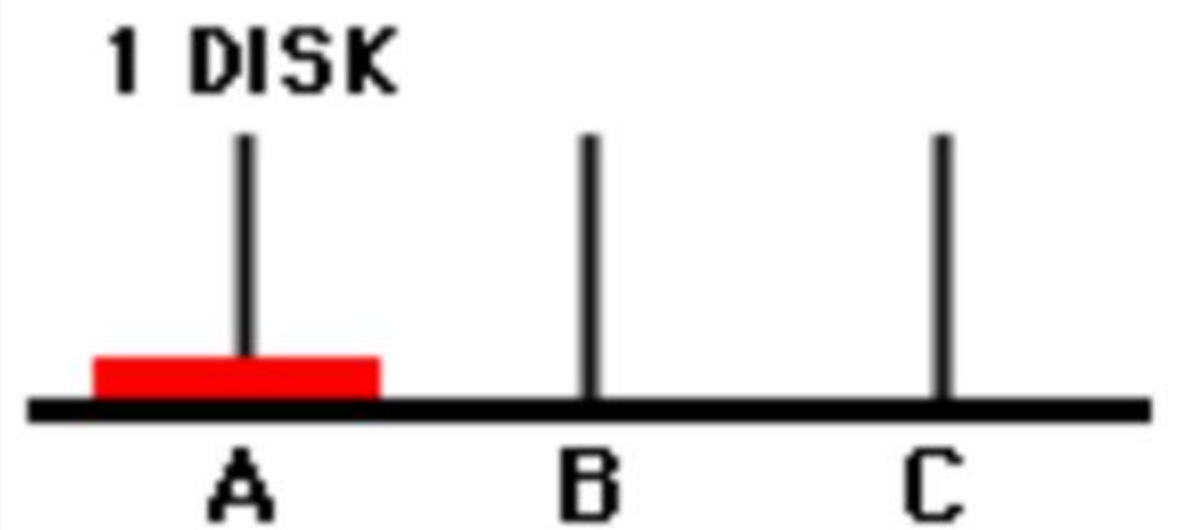
- Is a mathematical puzzle which consists of three tower *pegs* and more than one rings;
- These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one.
- There are other variations of puzzle where the number of disks increase, but the tower count remains the same.
- Tower of hanoi puzzle with **n** disks can be solved in minimum $\mathbf{2^n - 1}$ steps.



RULES

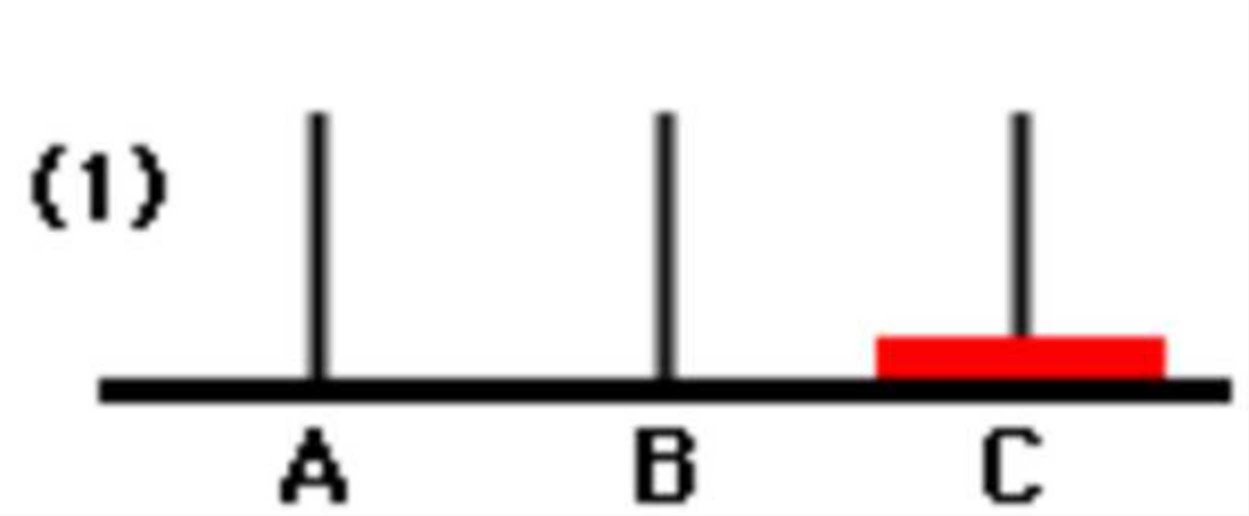
- The mission is to move all the disks to some another tower without violating the sequence of arrangement.
- Rules to be followed for Tower of Hanoi
  - Only one disk can be moved among the towers at any given time.
  - Only the "top" disk can be removed.
  - No large disk can sit over a small disk.

TOWERS OF HANOI WITH 1 DISK

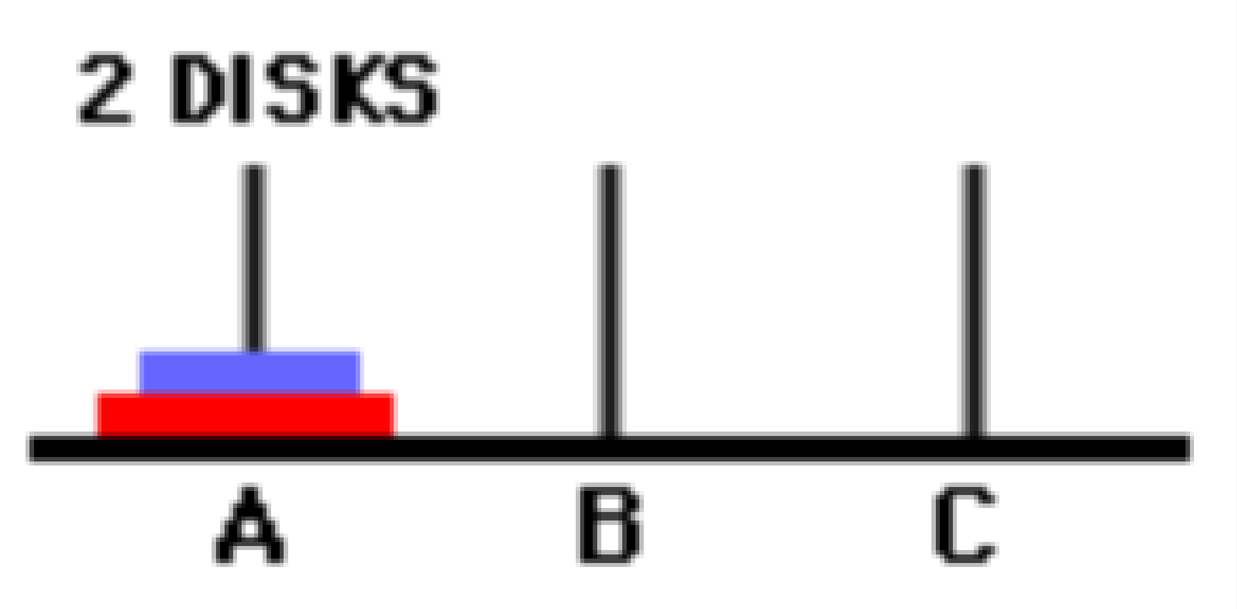


A – Source ,
C – Destination ,
B – Auxiliary

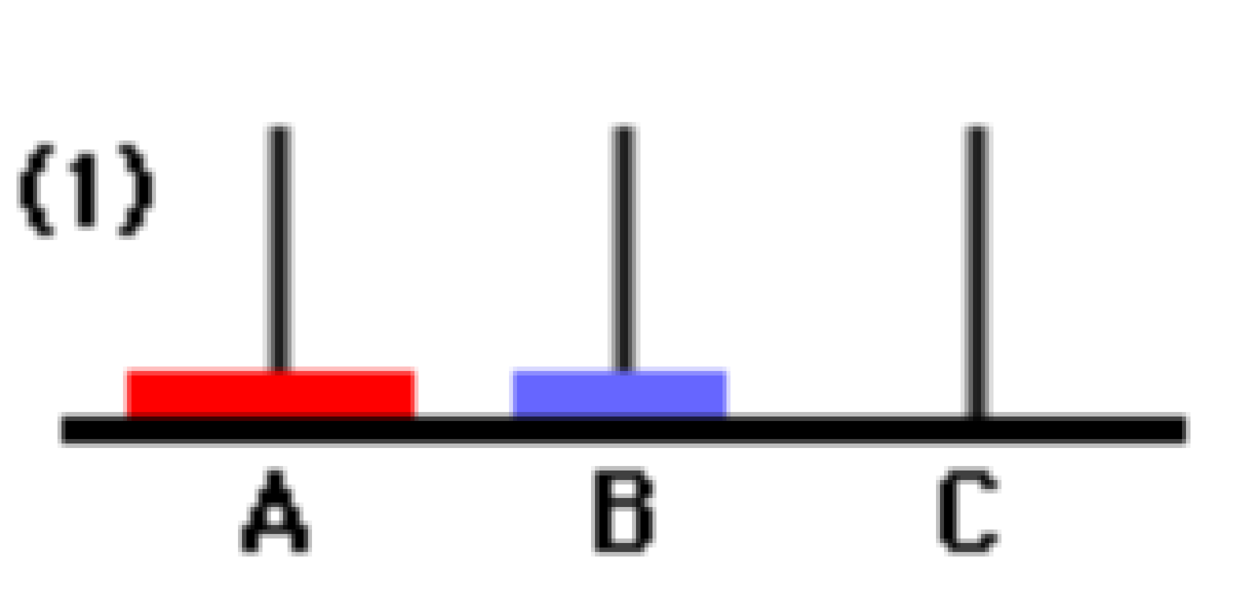
- **Move 1: Move disk 1 to post C**



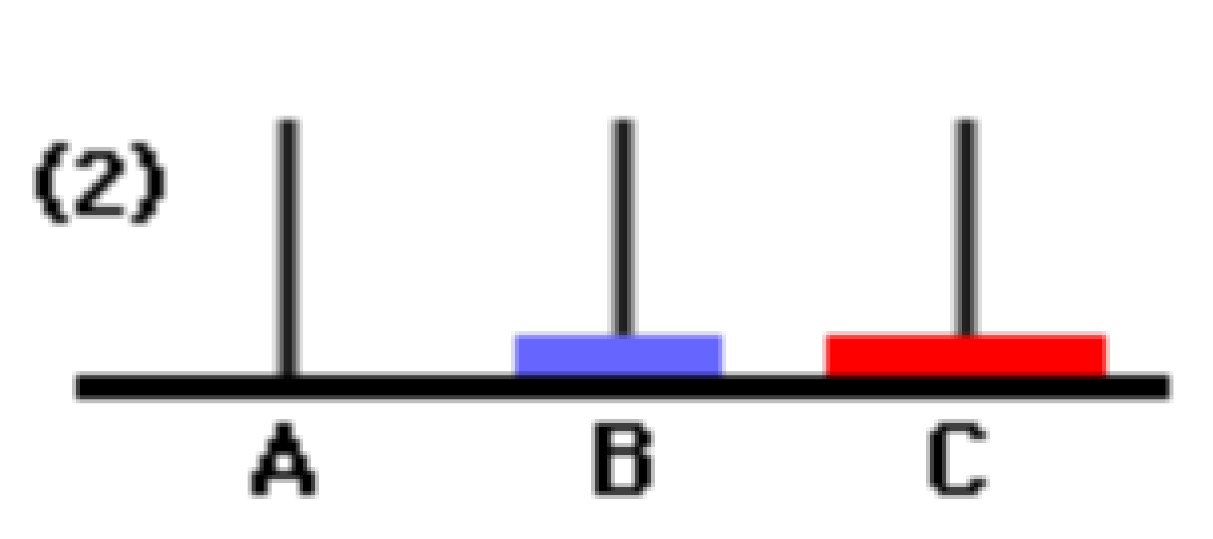
TOWERS OF HANOI WITH 2 DISK



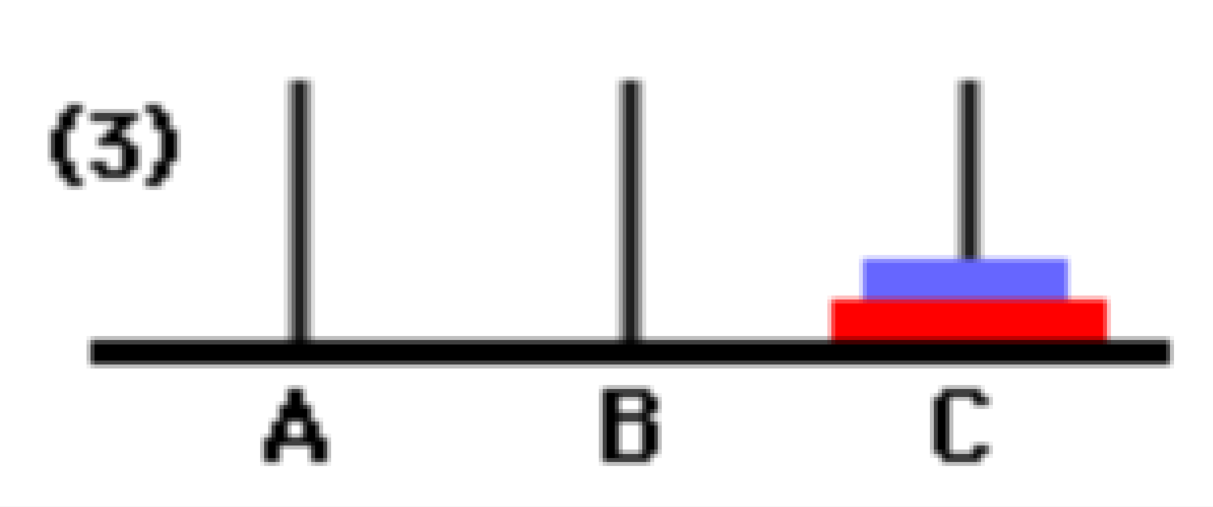
Move 1: Move disk 2 to post B



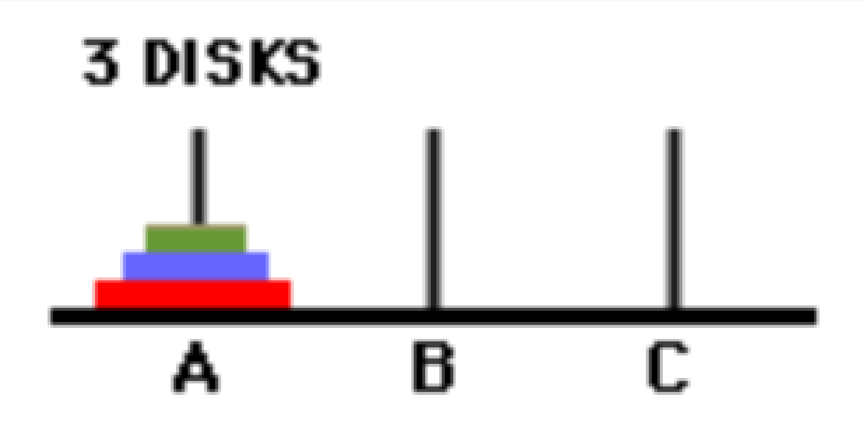
Move 2: Move disk 1 to post C

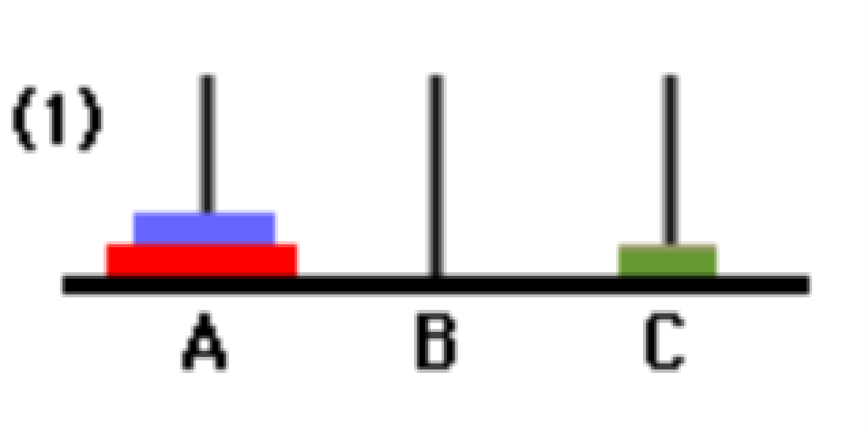


Move 3: Move disk 2 to post C

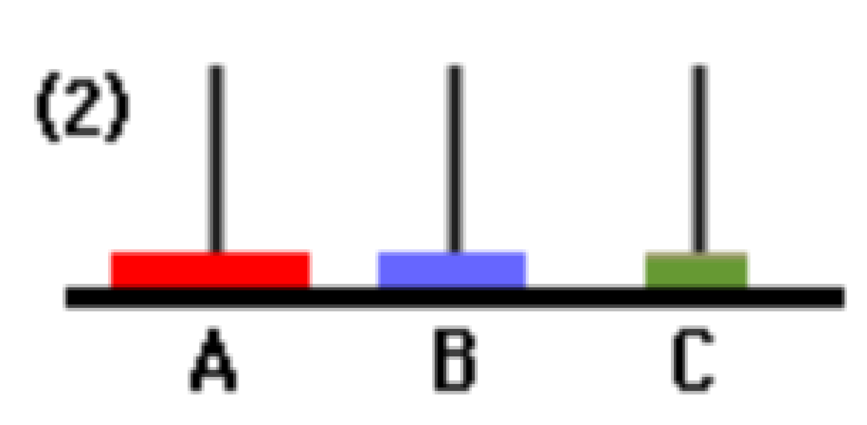


TOWERS OF HANOI WITH 3 DISKS

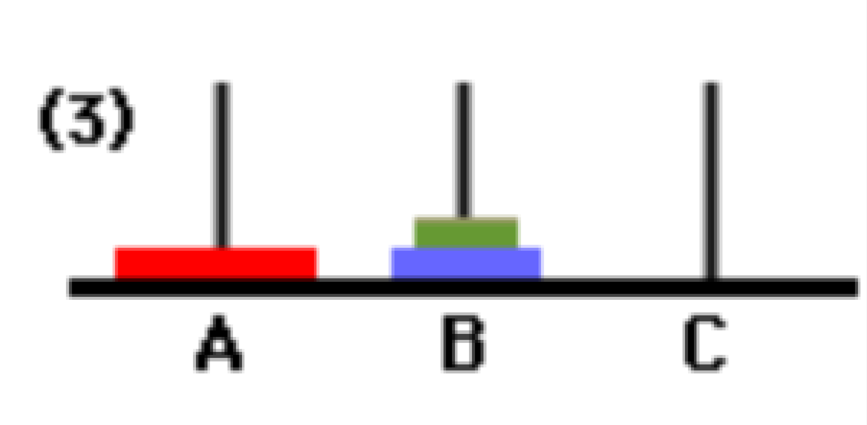




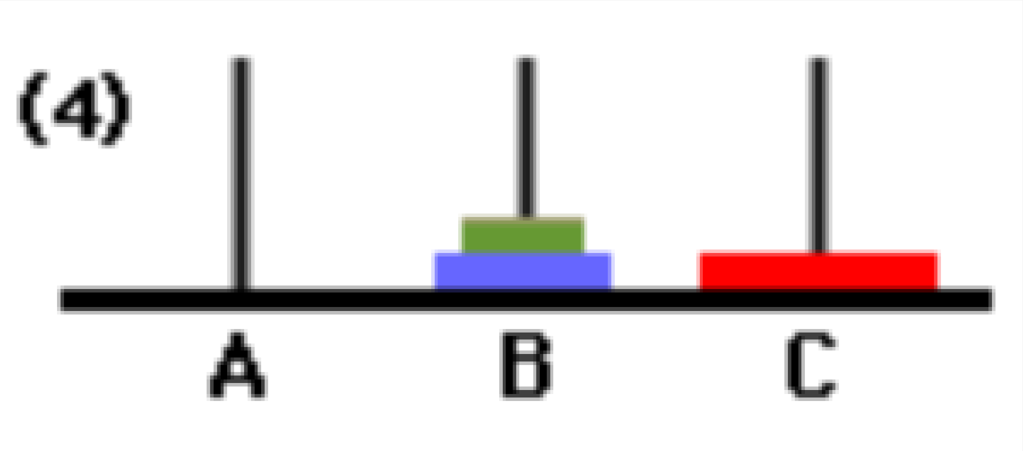
Move 1: Move disk 3 to post C



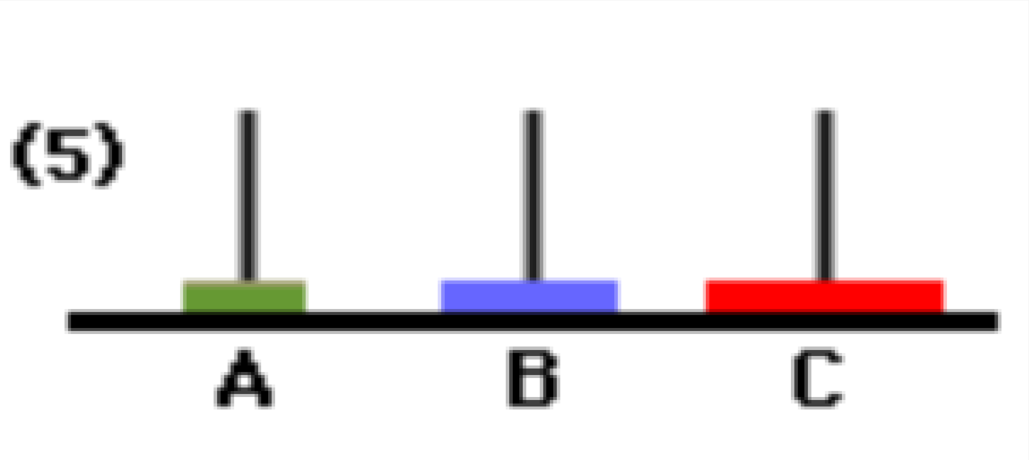
Move 2: Move disk 2 to post B



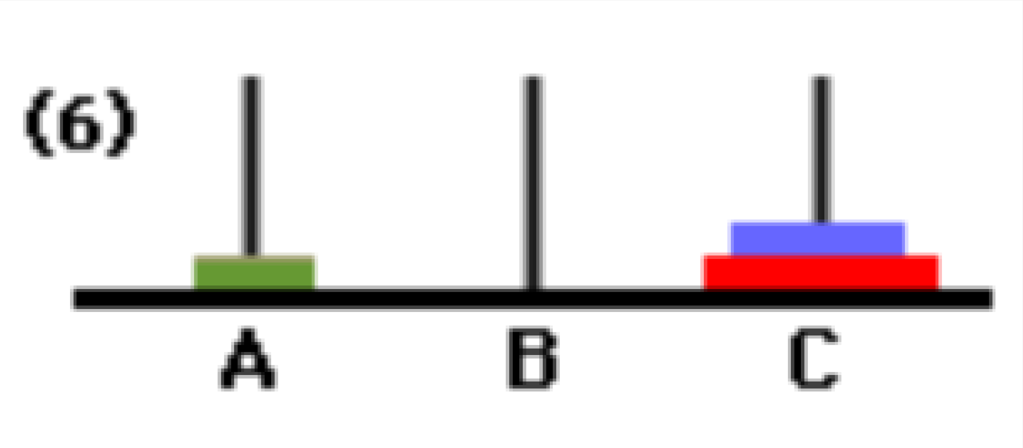
Move 3: Move disk 3 to post B



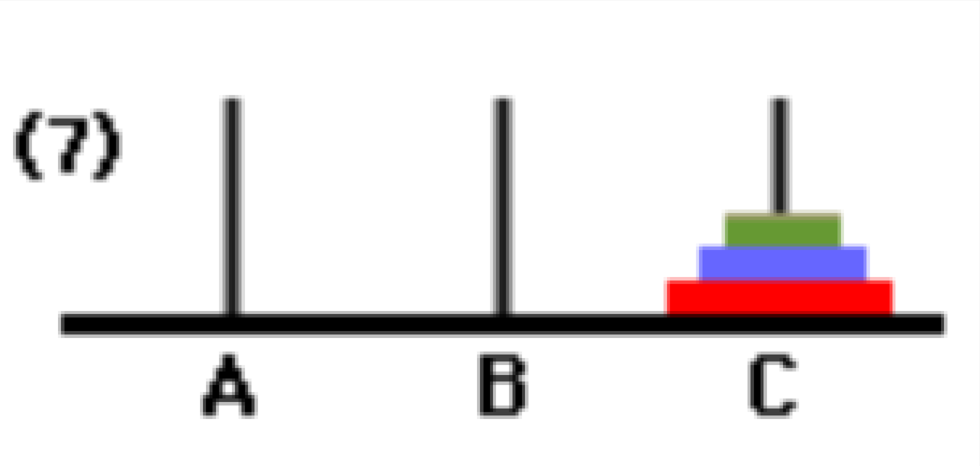
Move 4: Move disk 1 to post C



Move 5: Move disk 3 to post A

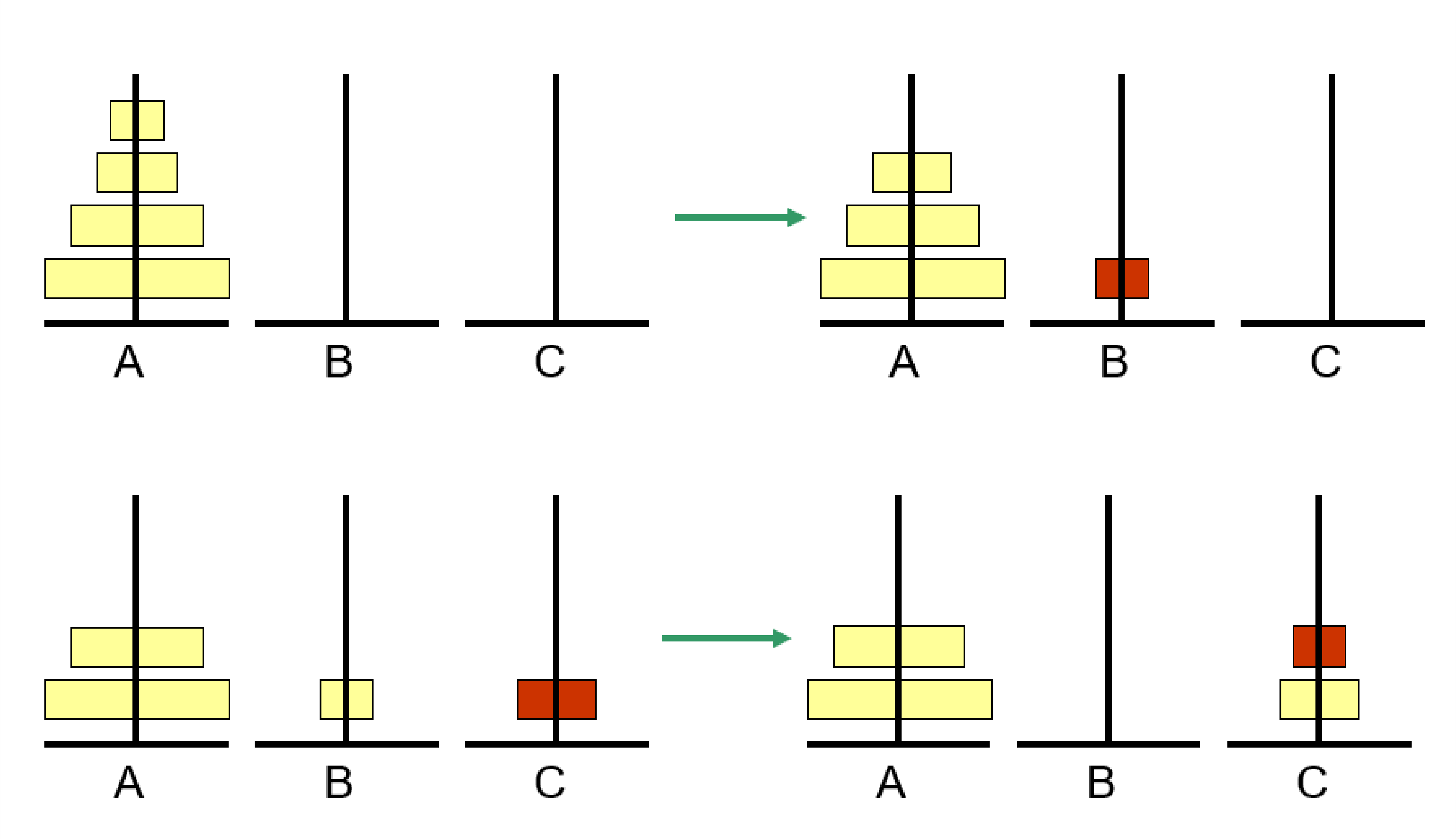


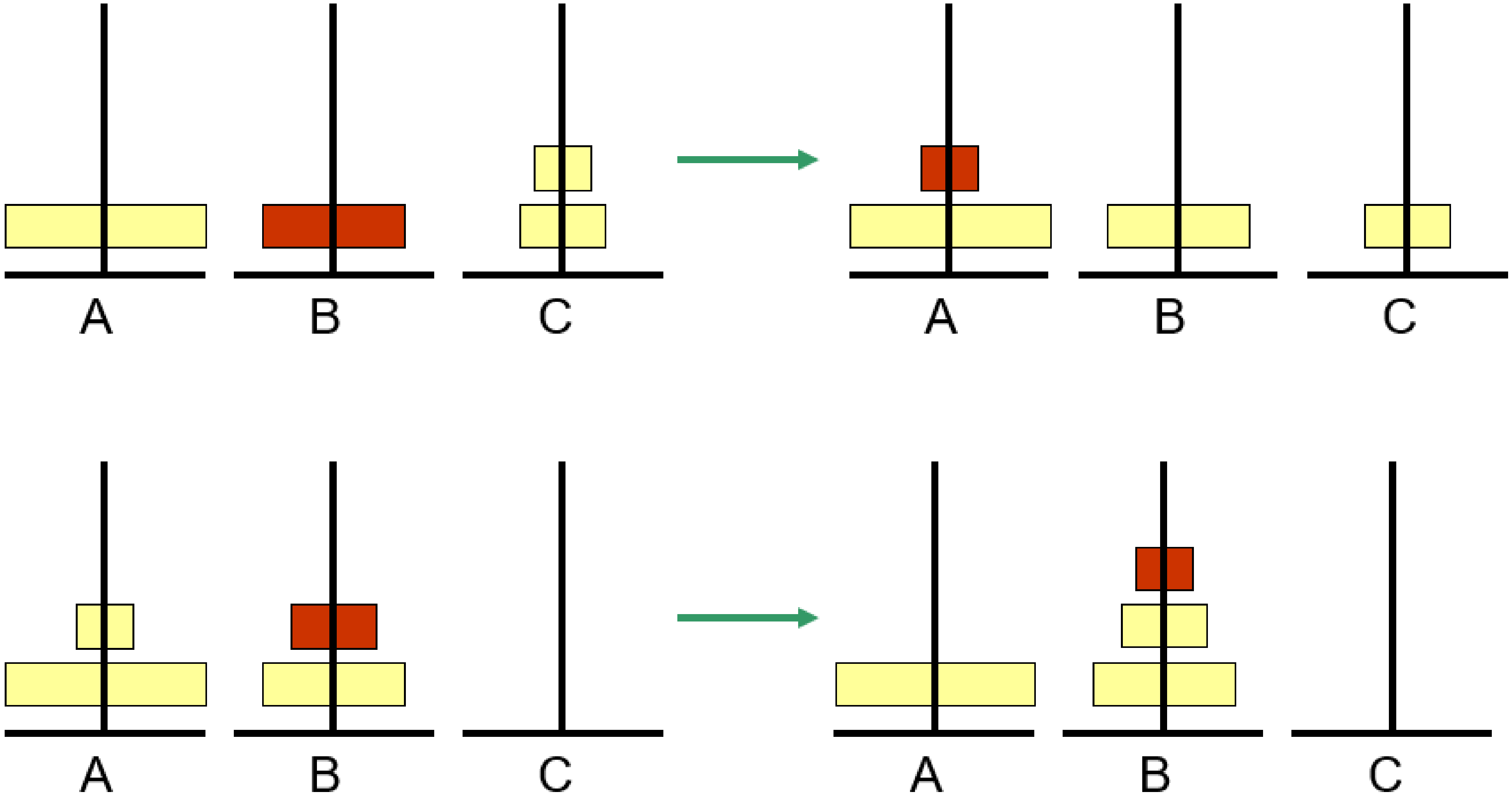
Move 6: Move disk 2 to post C

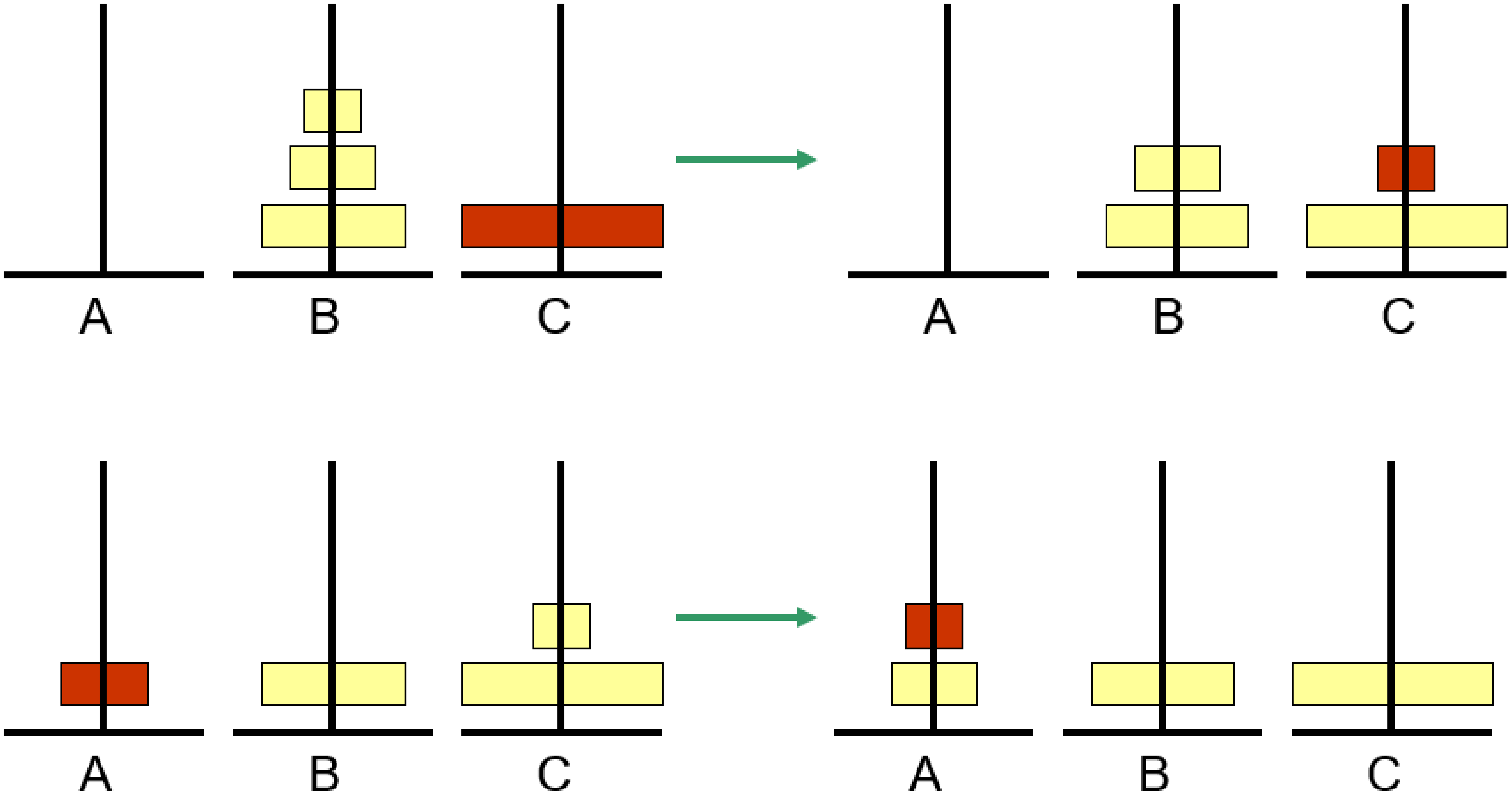


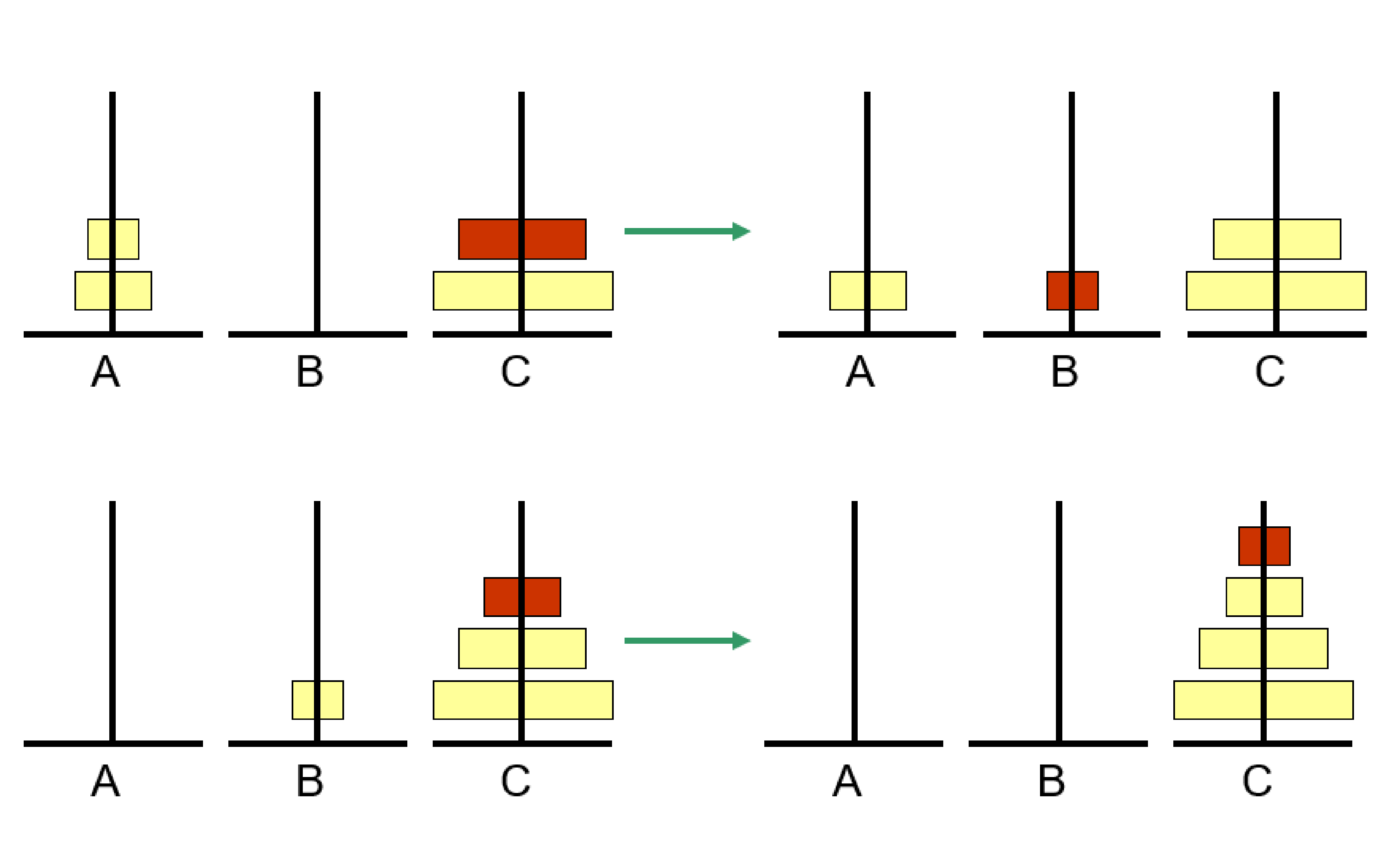
Move 7: Move disk 3 to post C

TOWERS OF HANOI WITH 4 DISKS




A
B
C
A
B
C
A
B
C
A
B
C


A
B
C
A
B
C
A
B
C
A
B
C


A
B
C
A
B
C
A
B
C
A
B
C

IMPLEMENTATION

```
int main()
{
        int n;
        printf("Enter the number of disks");
        scanf("%d",&n);
        printf("The disk movements are");
        tower(n,'A','C','B');
        return 0;
}
void tower(int n, char sp,char dp,char ap)
{
        if (n==1)
        {
                printf(Moving disk from %c to %c\n",sp,dp);
        }
        tower(n-1,sp,ap,dp);
        printf("Move disk  %d from %c to %c\n",n,sp,dp);
        tower(n-1,ap,dp,sp);
}
```

n = 3

$T(3, \overset{s}{A}, \overset{d}{C}, \overset{a}{B});$

n == 1 ✗

$T(2, \overset{s}{A}, \overset{d}{B}, \overset{a}{C})$

↓

$T(2, \overset{s}{A}, \overset{d}{B}, \overset{a}{C})$

n > 1 ✗

$T(1, \overset{s}{A}, \overset{d}{C}, \overset{a}{B})$

↓

n == 1 ✓

① Move 1 from A to C

return

② Move 2 from A to B

$T(1, \overset{s}{C}, \overset{d}{B}, \overset{a}{A})$

n == 1 ✓

③ Move 1 from C, to B.

return

④ Move disk 3 from A to C

$T(2, \overset{s}{B}, \overset{d}{C}, \overset{a}{A})$

n == 1 ✗

$T(1, \overset{s}{B}, \overset{d}{A}, \overset{a}{C})$

n == 1 ✓

⑤ Move 1 from B to

return

↓

⑥ Move 2

from B

$T(1, \overset{s}{A}, \overset{d}{C}, \overset{a}{B}$

↓

n == 1 ✓

⑦ Move 1 from

A to C

return

S = A
A = B
D = C

3	A	C	B
n	s	d	a

2	A	B	C
3	A	C	B

1	A	C	B
~~1~~	~~B~~	~~A~~	~~C~~
2	B	C	A
~~1~~	~~C~~	~~B~~	~~A~~
~~1~~	~~A~~	~~C~~	~~B~~
2	A	B	C
3	A	C	B

Moves

A → C
A → B
C → B.
A → C
B → A
B → C
A → C.

A → C
A → B
C → B
A → C.
B → A
B → C
A → C