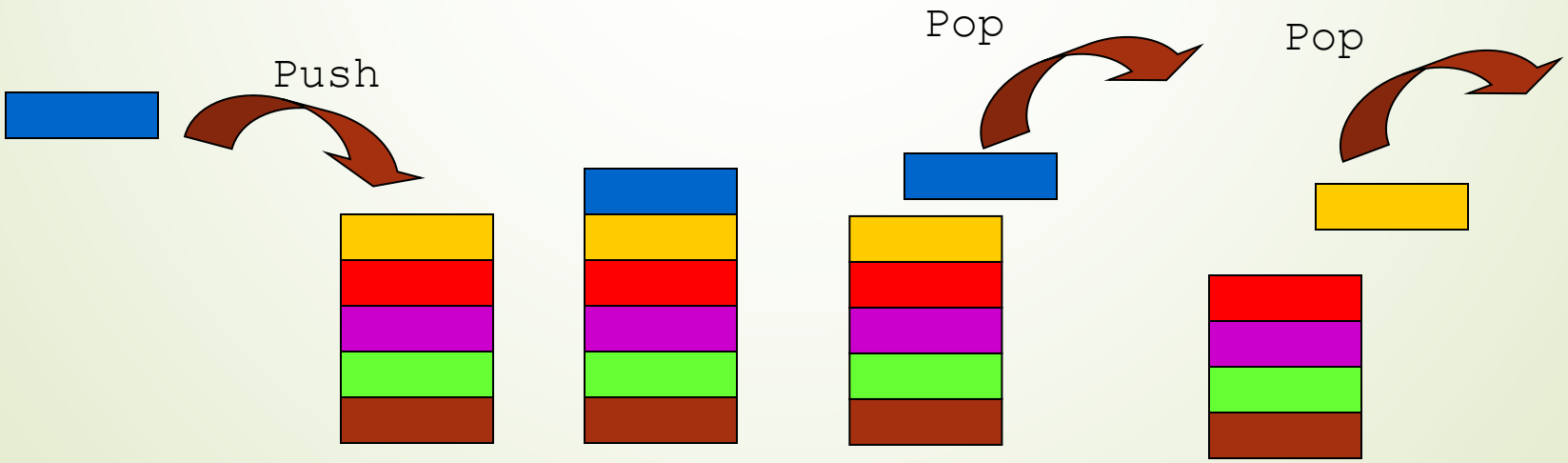
# Stack

- There are certain frequent situations in computer science when one wants to restrict insertion and deletions so that they can take place only at the beginning or at the end not in the middle.
  - **Stack**
  - **Queue**

# Stack

- A Stack is a list of elements in which an element may be inserted or deleted only at one end, call top of the Stack

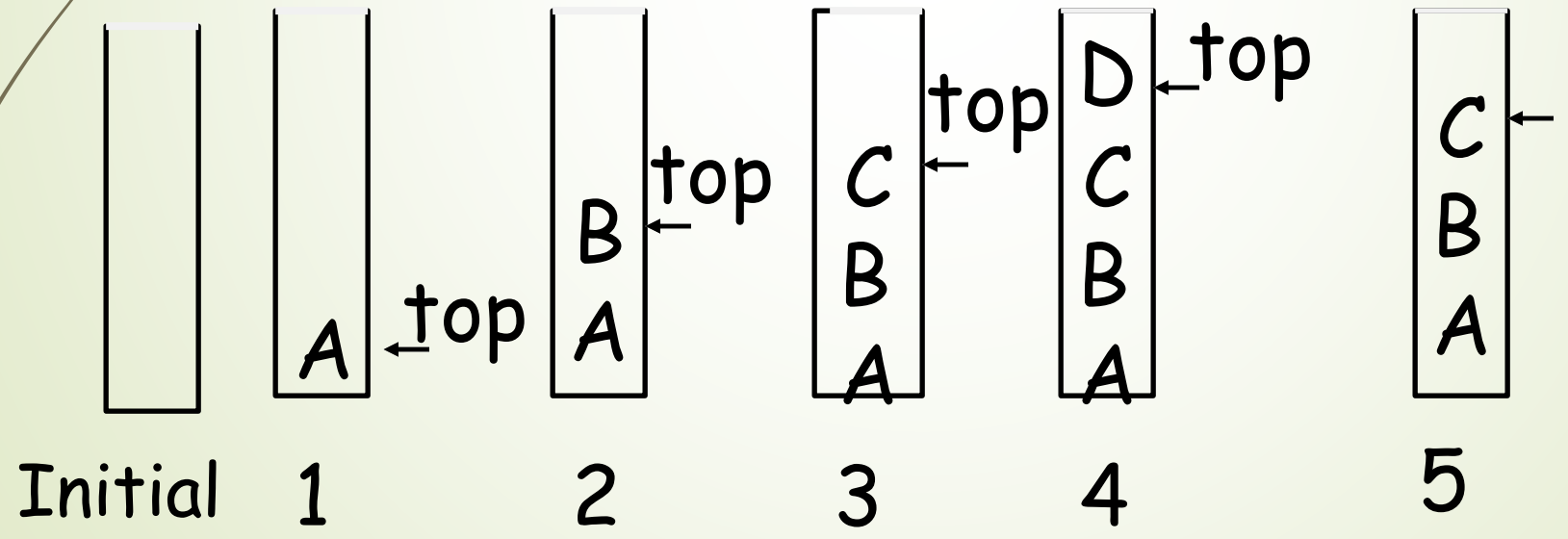
- Two basic operations are associated with Stack
  - Push : Insert an element into a stack
  - Pop : Delete an element from a stack



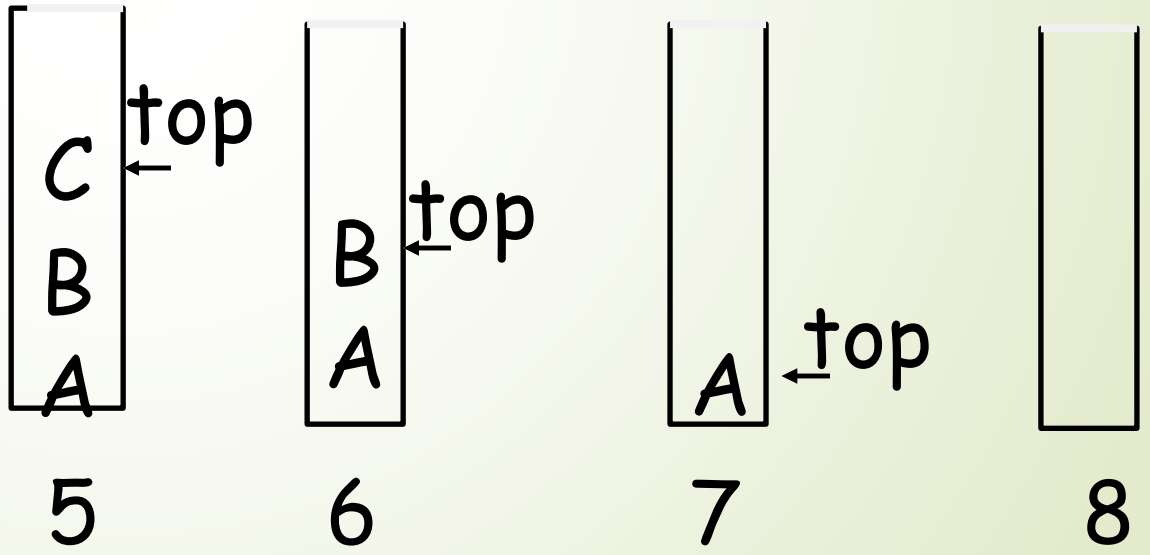
# Stack

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT= LIFO
- It means: the last element inserted is the first one to be removed

# Last In First Out

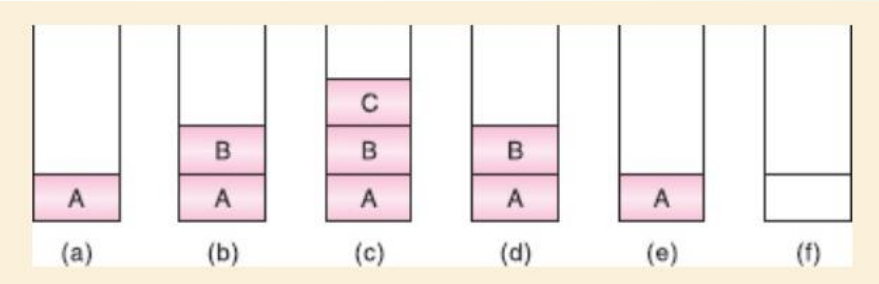


# Last In First Out



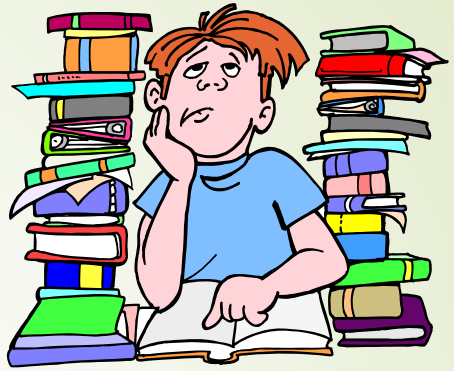
# Applications

- For postponed decisions
  - To indicate order of processing of data when certain steps of processing must be postponed until some other conditions are fulfilled




- Examples
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another

# Stack ADT

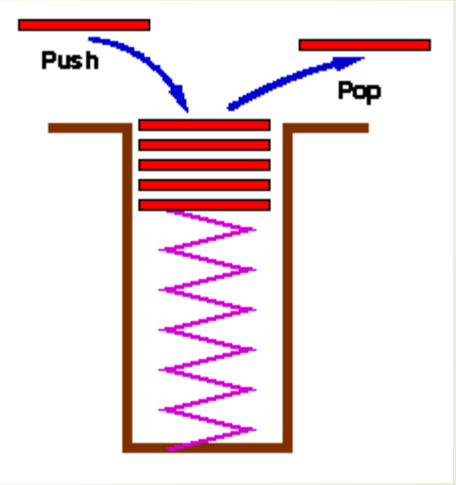


**Objects:**

A finite sequence of nodes

**Operations:**

- Create
- Push: Insert element at top
- Top: Return top element
- Pop: Remove and return top element
- IsEmpty: test for emptyness



# Stack ADT: Underflow

- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an Underflow Exception

# Representation of Stack

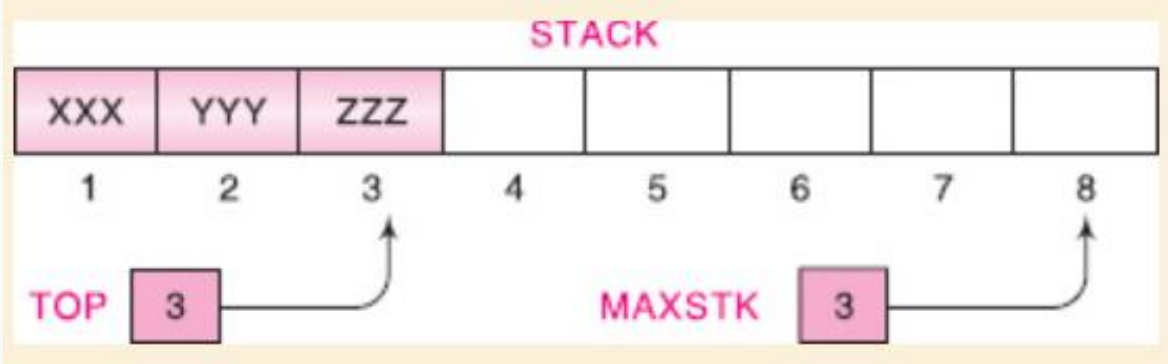
Stack can be represented in two different ways :

1 Linear ARRAY

2 One-way Linked list

# Array Representation

- Stack(TOP, MAXSTK)



# Push

PUSH(STACK, TOP, MAXSTK, ITEM)

1. [Stack already filled?]
If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

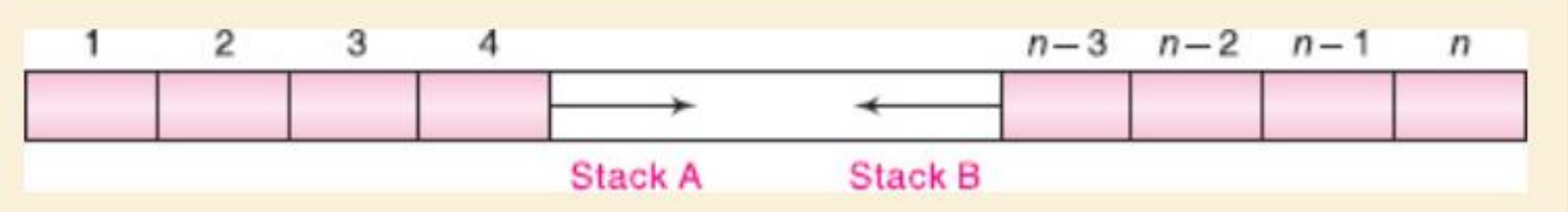
# Pop

POP(STACK, TOP, ITEM)

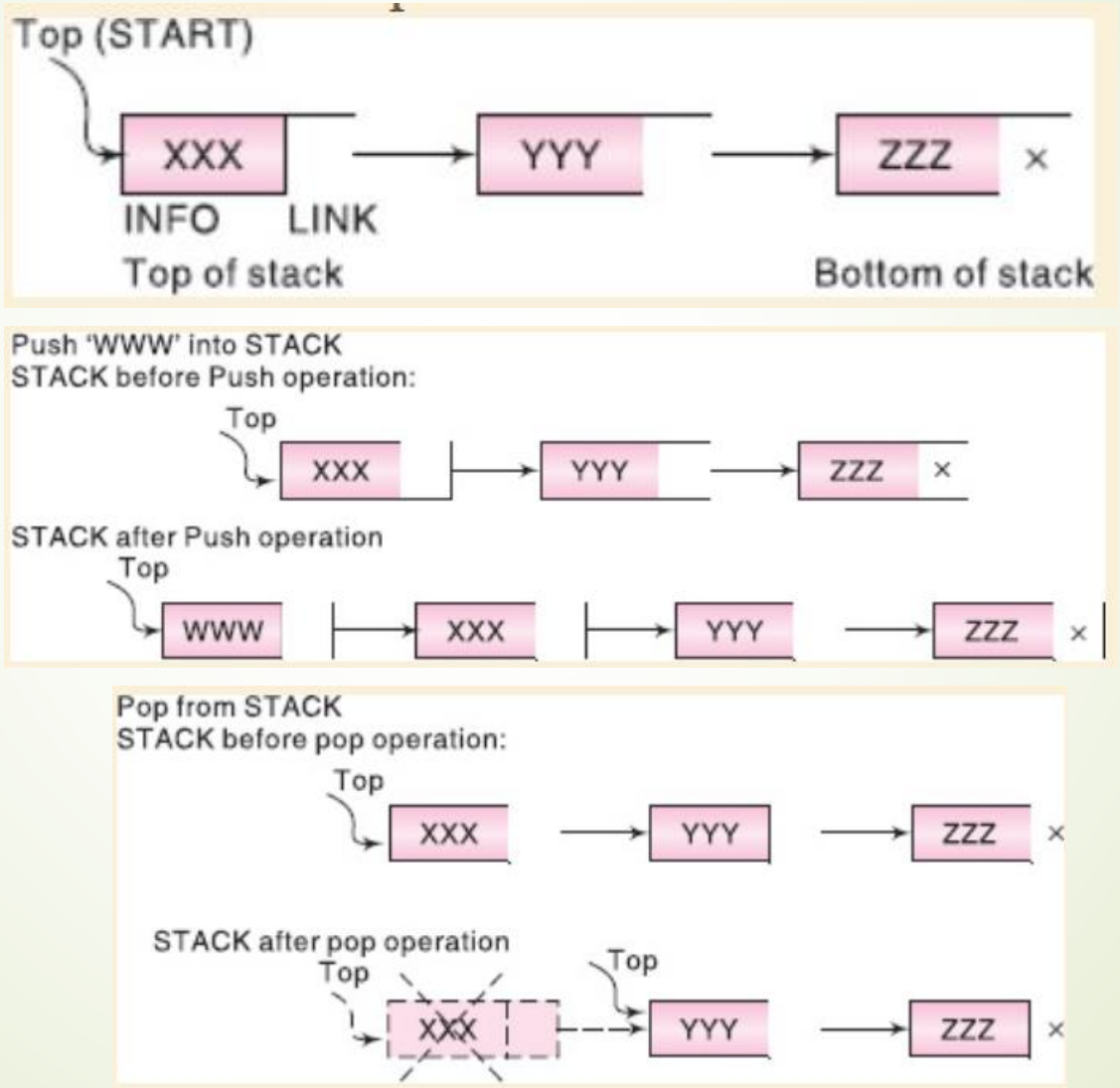
1. [Stack has an item to be removed?]
If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP – 1. [Decreases TOP by 1.]
4. Return.

# Minimizing Overflow

- Number of overflow depends upon size of stack
- Time-space trade-off between number of overflow and stack space



# Linked List Representation



# Push

PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)

1. [Available space?] If AVAIL = NULL, then Write OVERFLOW and Exit
2. [Remove first node from AVAIL list]
Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM [ Copies ITEM into new node]
4. Set LINK[NEW] := TOP [New node points to the original top node in the stack]
5. Set TOP = NEW [Reset TOP to point to the new node at the top of the stack]
6. Exit.

# Pop

POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)

**1**. [Stack has an item to be removed?]

IF TOP = NULL then Write: UNDERFLOW and Exit.

**2**. Set ITEM := INFO[TOP] [Copies the top element of stack into ITEM ]

**3**. Set TEMP := TOP and TOP = LINK[TOP]

[Remember the old value of the TOP pointer in TEMP and reset TOP to point to the next element in the stack ]

**4**. [Return deleted node to the AVAIL list]

Set LINK[TEMP] = AVAIL and AVAIL = TEMP.

**5**. Exit.

# Example

(i) Push BBB (ii) Pop (iii) Pop (iv) Push MMM

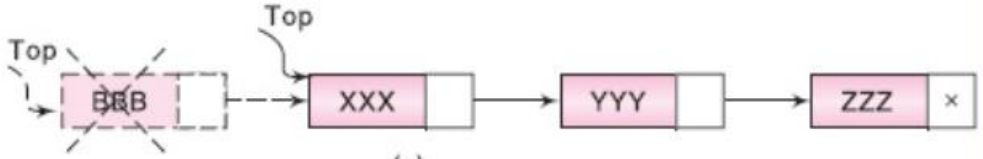
Original linked stack:


(a)

(i) Push BBB

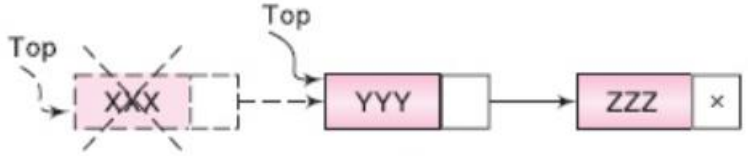

(b)

(ii) Pop




(c)

(iii) Pop




(d)

(iv) Push MMM


(e)

# Application: Polish Notation

- Let **Q** be an arithmetic expression involving constant and operations
- Find the value of **Q** using reverse Polish (**Postfix**) Notation

# Arithmetic Expression

- Evaluate the following parenthesis-free arithmetic expression

**2 î 3 + 5 * 2 î 2 – 12 / 6**

Evaluate the exponentiation to obtain

**8 + 5 * 4 – 12 /6**

Evaluate Multiplication and Division

**8 + 20 – 2**

Evaluate Addition and Subtraction

**26**

# Polish Notation

- **Infix notation** [Operator symbol is placed between two Operand]

  A + B , C – D , E * F , G /H

  (A + B) * C and A + (B*C)

- **Polish Notation** [Operator symbol is placed before its operand]

  +AB, -CD, *EF , /GH

Polish Notations are frequently called **Prefix**

# Polish Notation

- Infix expression to Polish Notation

[ ] to indicate a partial translation

(A+B)*C = [+AB]*C = *+ABC

A+(B*C) = A+[*BC] = +A*BC

(A+B)/(C-D) = [+AB]/[-CD] = /+AB-CD

# Polish Notation

- The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operand in the expression.
- **One never needs parenthesis when writing expression in Polish notations**

# Reverse Polish Notation

- Operator symbol is placed after its two operand

  AB+, CD-, EF*, GC/

**One never needs parenthesis to determine the order of the operation in any arithmetic expression written in reverse Polish notation.**

**Also known as Postfix (or Suffix) notation**

- Computer usually evaluates an arithmetic expression written in infix notation in two steps:
- First Step: Converts the **expression to Postfix notation**
- Second Step: **Evaluates the Postfix expression.**

# Evaluation of Postfix Expression

- Algorithm to find the **Value** of an arithmetic expression **P** Written in **Postfix**

1 Add a right parenthesis ")" at the end of P. [This act as delimiter]

2 Scan P from left to right and repeat Steps 3 and 4 for each element of P until the delimiter ")" is encountered

3 If an operand is encountered, put it on STACK

4 If an operator $\otimes$ is encountered, then

(a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element

(b) Evaluate B $\otimes$ A

(c ) Place the result of (b) on STACK

5 Set **Value** equal to the top element of STACK

6 Exit

# Example

- P = 5, 6, 2, + , *, 12, 4, /, - [Postfix]
- Q = 5 * ( 6 + 2) - 12 / 4 [Infix]

- P:

| **5,** | **6,** | **2,** | **+,** | ***,** | **12,** | **4,** | **/,** | **-,** | **)** |
|---|---|---|---|---|---|---|---|---|---|
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |

5, 6, 2, +, *, 12, 4, /, -, )

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

| Symbol Scanned | | STACK | | |
|---|---|---|---|---|
| (1) | 5 | 5 | | |
| (2) | 6 | 5, | 6 | |
| (3) | 2 | 5, | 6, | 2 |
| (4) | + | 5, | 8 | |
| (5) | * | 40 | | |
| (6) | 12 | 40, | 12 | |
| (7) | 4 | 40, | 12, | 4 |
| (8) | / | 40, | 3 | |
| (9) | - | 37 | | |
| (10) | ) | | | |

# Infix to Postfix

- Q is an arithmetic expression written in infix notation
- ^ , * , / , + , -
- Three level of precedence

# Infix to Postfix

- Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix notation

1 Push "(" onto STACK and ")" to the end of Q

2 Scan Q from Left to Right and Repeat Steps 3 to 6 for each element of Q until the STACK is empty

3 If an operand is encountered, add it to P

4 If a left parenthesis is encountered, push it onto STACK

5 If an operator $\otimes$ is encountered, then:

(a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has same precedence as or higher precedence than $\otimes$ .

(b) Add $\otimes$ to STACK

6 If a right parenthesis is encountered, then

(a) Repeatedly pop from the STACK and add to P each operator (on top of STACK) until a left parenthesis is encountered.

(b) Remove the left parenthesis. [Do not add it to P]

7 Exit

# Example

- Q : A + ( B * C – ( D / E î F ) * G ) * H

| A | + | ( | B | * | C | - | ( | D | / | E | î | F | ) | * | G | ) | * | H | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | | | | 20 |

| A | + | ( | B | * | C | - | ( | D | / | E | î | F | ) | * | G | ) | * | H | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | | | | 20 |

| Symbol Scanned | | STACK | Expression P |
|---|---|---|---|
| 1 | A | ( | A |
| 2 | + | ( + | A |
| 3 | ( | ( + ( | A |
| 4 | B | ( + ( | A B |
| 5 | * | ( + ( * | A B |
| 6 | C | ( + ( * | A B C |
| 7 | - | ( + ( - | A B C * |
| 8 | ( | ( + ( - ( | A B C * |

| A | + | ( | B | * | C | - | ( | D | / | E | ↑ | F | ) | * | G | ) | * | H | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | | | | 20 |

| Symbol Scanned | STACK | Expression P |
|---|---|---|
| 8 ( | ( + ( - ( | A B C * |
| 9 D | ( + ( - ( | A B C * D |
| 10 / | ( + ( - ( / | A B C * D |
| 11 E | ( + ( - ( / | A B C * D E |
| 12 ↑ | ( + ( - ( / ↑ | A B C * D E |
| 13 F | ( + ( - ( / ↑ | A B C * D E F |
| 14 ) | ( + ( - | A B C * D E F ↑ / |
| 15 * | ( + ( - * | A B C * D E F ↑ / |

| A | + | ( | B | * | C | - | ( | D | / | E | ↑ | F | ) | * | G | ) | * | H | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | | | | 20 |

| Symbol Scanned | STACK | Expression P |
|---|---|---|
| 15 * | ( + ( - * | A B C * D E F ↑ / |
| 16 G | ( + ( - * | A B C * D E F ↑ / G |
| 17 ) | ( + | A B C * D E F ↑ / G * - |
| 18 * | ( + * | A B C * D E F ↑ / G * - |
| 19 H | ( + * | A B C * D E F ↑ / G * - H |
| 20 ) | | A B C * D E F ↑ / G * - H * + |

# Application: Quick Sort



Quick sort is an algorithm of the divide-and-conquer type

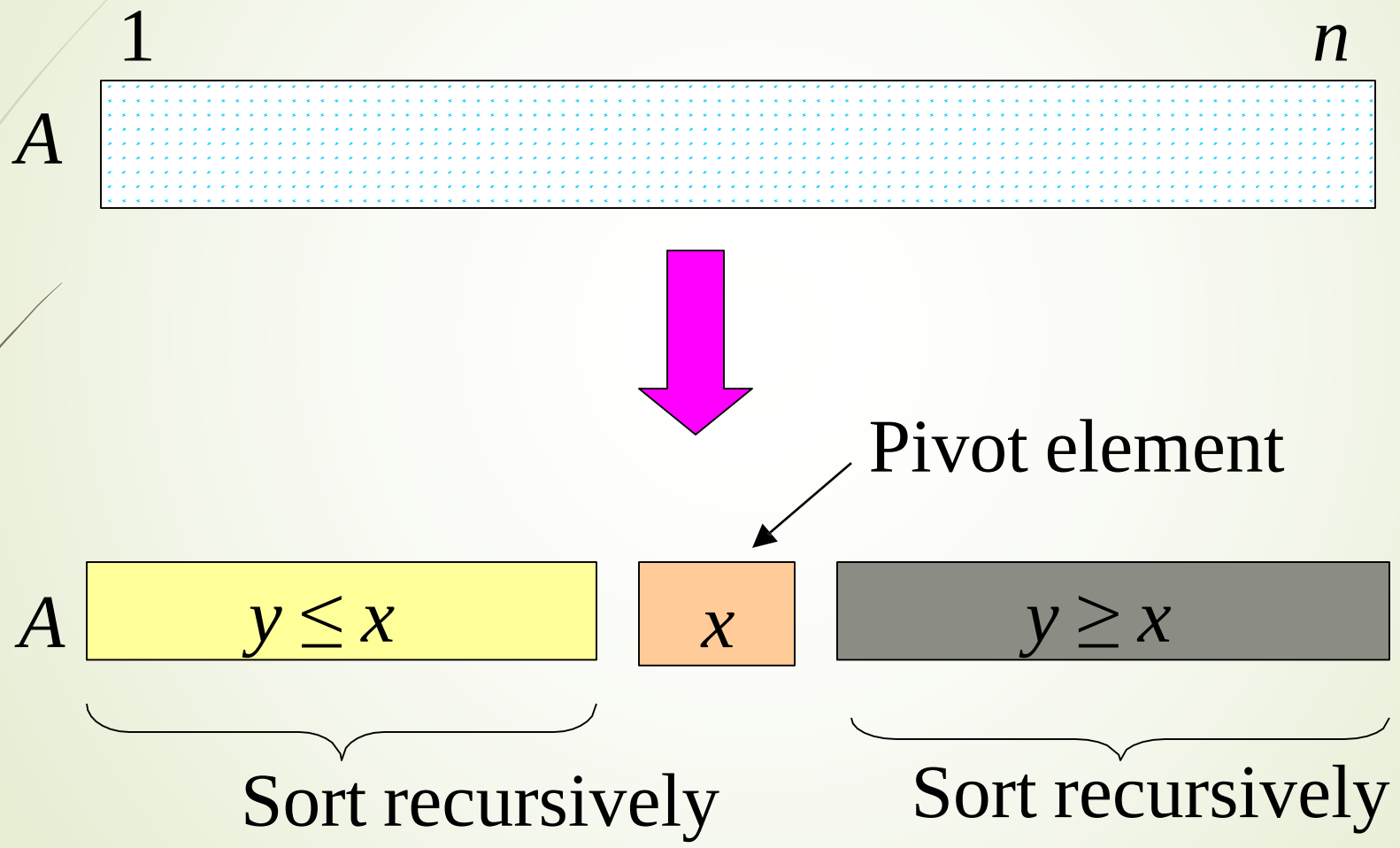
The problem of sorting a set is reduced to the problem of sorting two smaller sets.

# Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element (first) to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

# Quick Sort Approach



# Quick Sort: divide the list

Select the first number FIRST as pivot.

Begining with the last number in the list, scan from right to left, comparing each number with FIRST and stopping at the first number less than FIRST. Then interchange the two number.

Starting from the interchanged number, scan the list from left to right, comparing each number with FIRST and stopping at the first number greater than FIRST. Then interchange the two number.

Repeat the above process until FIRST is settled to its final place

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

First Sublist

Second SubList

# Quick Sort Algorithm

- **Input**: Unsorted sub-array A[first..last]
- **Output**: Sorted sub-array A[first..last]

- **QUICKSORT** (A, first, Last)
  - **if** first < last

**then** **loc**← **PARTITION**(*A*, first, *last*)

**QUICKSORT** (A, first, loc-1)

**QUICKSORT** (A, loc+1, last)

# Quicksort: use of stacks

- Two stacks LOWER and UPPER

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

LOWER: 1 UPPER: 12

LOWER: (empty) UPPER: (empty)

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

LOWER: 1, 6 UPPER: 4, 12

LOWER: 1 UPPER: 4

66, 77, 60, 88, 55, 90, 99

LOWER: 1, 6 UPPER: 4, 10

QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
   **(a)** Repeat while A[LOC) ≤ A[RIGHT] and LOC ≠ RIGHT:
   RIGHT := RIGHT – 1.
   [End of loop.]
   **(b)** If LOC = RIGHT, then: Return.
   **(c)** If A[LOC] > A[RIGHT], then:
   **(i)** [Interchange A[LOC) and A[RIGHT].]
   TEMP := A[LOC), A[LOC] := A[RIGHT),
   A[RIGHT] := TEMP.
   **(ii)** Set LOC := RIGHT.
   **(iii)** Go to Step 3.
   [End of If structure.]

**3**. [Scan from left to right.]

**(a)** Repeat while A[LEFT] ≤ A[LOC) and LEFT ≠ LOC:

LEFT := LEFT + 1.

[End of loop.]

**(b)** If LOC = LEFT, then: Return.

**(c)** If A[LEFT] > A[LOC], then

**(i)** [Interchange A[LEFT] and A[LOC].]

TEMP := A[LOC], A[LOC] := A[LEFT],

A[LEFT] := TEMP.

**(ii)** Set LOC := LEFT.

**(iii)** Go to Step 2.

[End of If structure.]

(Quicksort) This algorithm sorts an array A with N elements.

**1.** [Initialize.] TOP := NULL.

**2.** [Push boundary values of A onto stacks when A has 2 or more elements.]
If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.

**3.** Repeat Steps 4 to 7 while TOP ≠ NULL.

**4.** [Pop sublist from stacks.]
Set BEG := LOWER[TOP], END := UPPER[TOP],
TOP := TOP – 1.

**5.** Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]

**6.** [Push left sublist onto stacks when it has 2 or more elements.]
If BEG < LOC – 1, then:
TOP := TOP + 1, LOWER[TOP] := BEG,
UPPER[TOP] = LOC – 1.
[End of If structure.]

**7.** [Push right sublist onto stacks when it has 2 or more elements.]
If LOC + 1 < END, then:
TOP := TOP + 1, LOWER[TOP] := LOC + 1,
UPPER[TOP] := END.
[End of If structure.]
[End of Step 3 loop.]

**8.** Exit.

# Complexity of Quicksort

- For simplicity, assume $n = 2^m$ so that $m = \log_2 n$
- Proper position of pivot always turns out to be middle of sub-array

| Pass | Number of sub-arrays | number of comparisons |
|---|---|---|
| 1 | 1 | n |
| 2 | 2 | 2*(n/2) |
| 3 | 4 | 4*(n/4) |
| . | | |
| . | | |
| m | $2^m$ | $2^m * (n/2^m)$ |

Total number of comparisons = $n*m = n\log_2 n$

Order of complexity = $O(n\log_2 n)$

In general, on average, number of comparisons = $1.386\ n\log_2 n$

In general, quicksort is the fastest sorting algorithm because of low overhead and average $O(n\log_2 n)$ complexity

# Worst and Best Case

- Worst case: if array is completely sorted or unsorted
  - Heavily unbalanced partition: n-1 and 0
  - $(n-1)+(n-2)+\ldots+2+1 = O(n^2)$
- Best case: if the size of two partitions are same
  - $O(n\log_2 n)$

# Choice of pivot element

- Median-of-the-three
  - Median of first, last and middle element
- Meansort
  - In first step, first or median of three technique is used
  - In subsequent steps, mean of sub-array is used as pivot
- Bsort
  - Middle element as pivot
- Divides the array evenly
- Quicksort is O(nlogn) even for sorted files
- Meansort and Bsort are recommended if the array is known to be almost sorted

# Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler etc.
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

# Recursion

- The process in which a function calls itself directly or indirectly is called recursion
- Corresponding function is called as recursive function
- Using recursive algorithm, certain problems can be solved quite easily.
- Solution of the bigger problem is expressed in terms of smaller problems.
- Well defined recursive function
  - There must be certain arguments, called base values, for which function does not refer itself (terminating condition)
  - Each time function does refer to itself, argument of the function must be more closer to the base values.

# Factorial function

- Factorial n: product of positive integers from 1 to n

$$n! = 1 \cdot 2 \cdot 3 \ldots (n-2)(n-1)n$$

- 0! = 1
- Recursive definition

$$n! = n \cdot (n \cdot 1)!$$

- Well defined recursive definition

If $n = 0$, then $n! = 1$.

If $n > 0$, then $n! = n \cdot (n - 1)!$

# 4!

(1) $4! = 4 \cdot 3!$

(2) $3! = 3 \cdot 2!$

(3) $2! = 2 \cdot 1!$

(4) $1! = 1 \cdot 0!$

(5) $0! = 1$

(6) $1! = 1 \cdot 1 = 1$

(7) $2! = 2 \cdot 1 = 2$

(8) $3! = 3 \cdot 2 = 6$

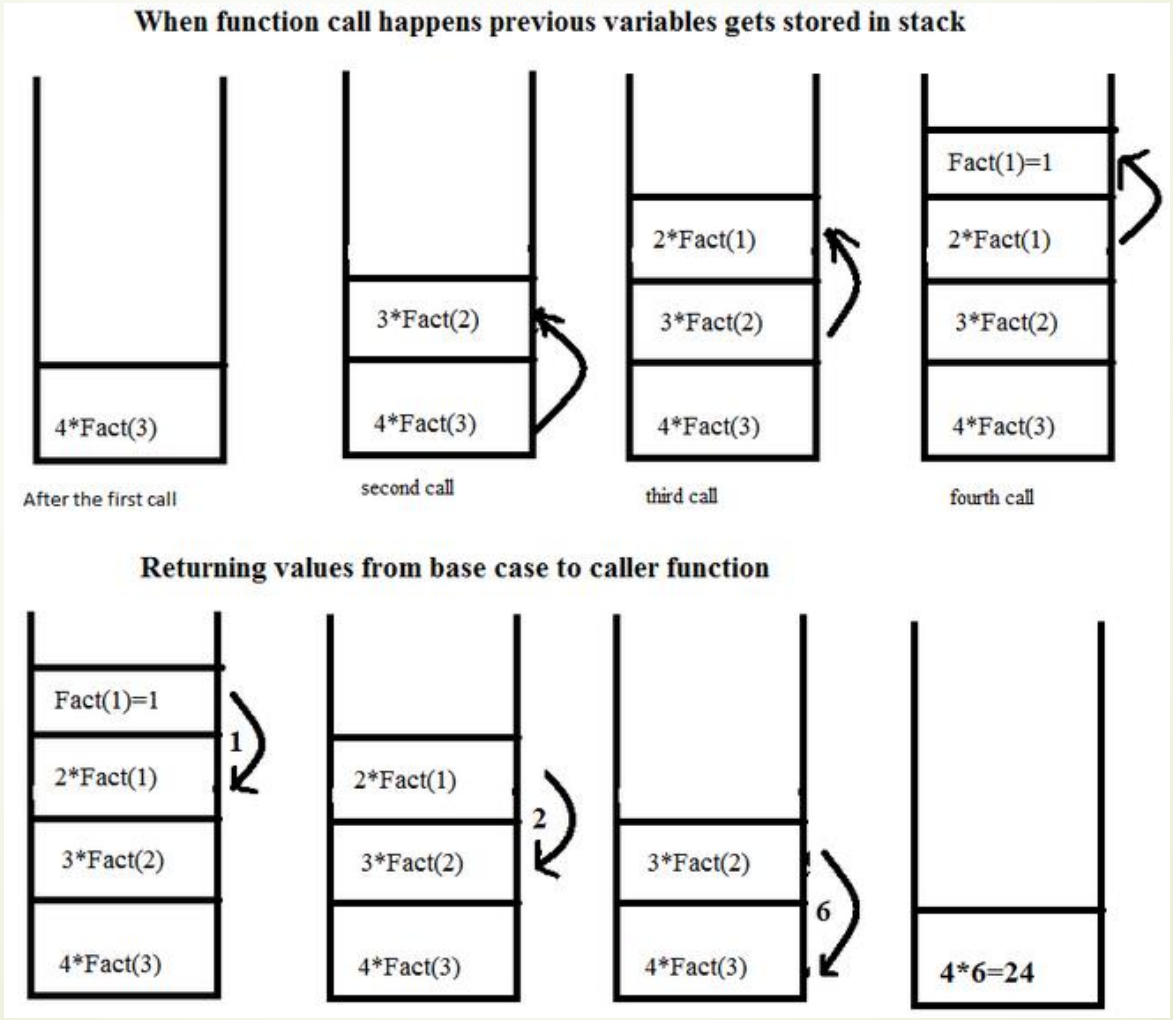
(9) $4! = 4 \cdot 6 = 24$

# Recursive function for factorial

FACTORIAL(FACT, N)

1. If N = 0, then: Set FACT := 1, and Return.
2. Call FACTORIAL(FACT, N – 1).
3. Set FACT := N*FACT.
4. Return.

# 4! using stack



# Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots.$$

$$F_0 = 0 \text{ and } F_1 = 1$$

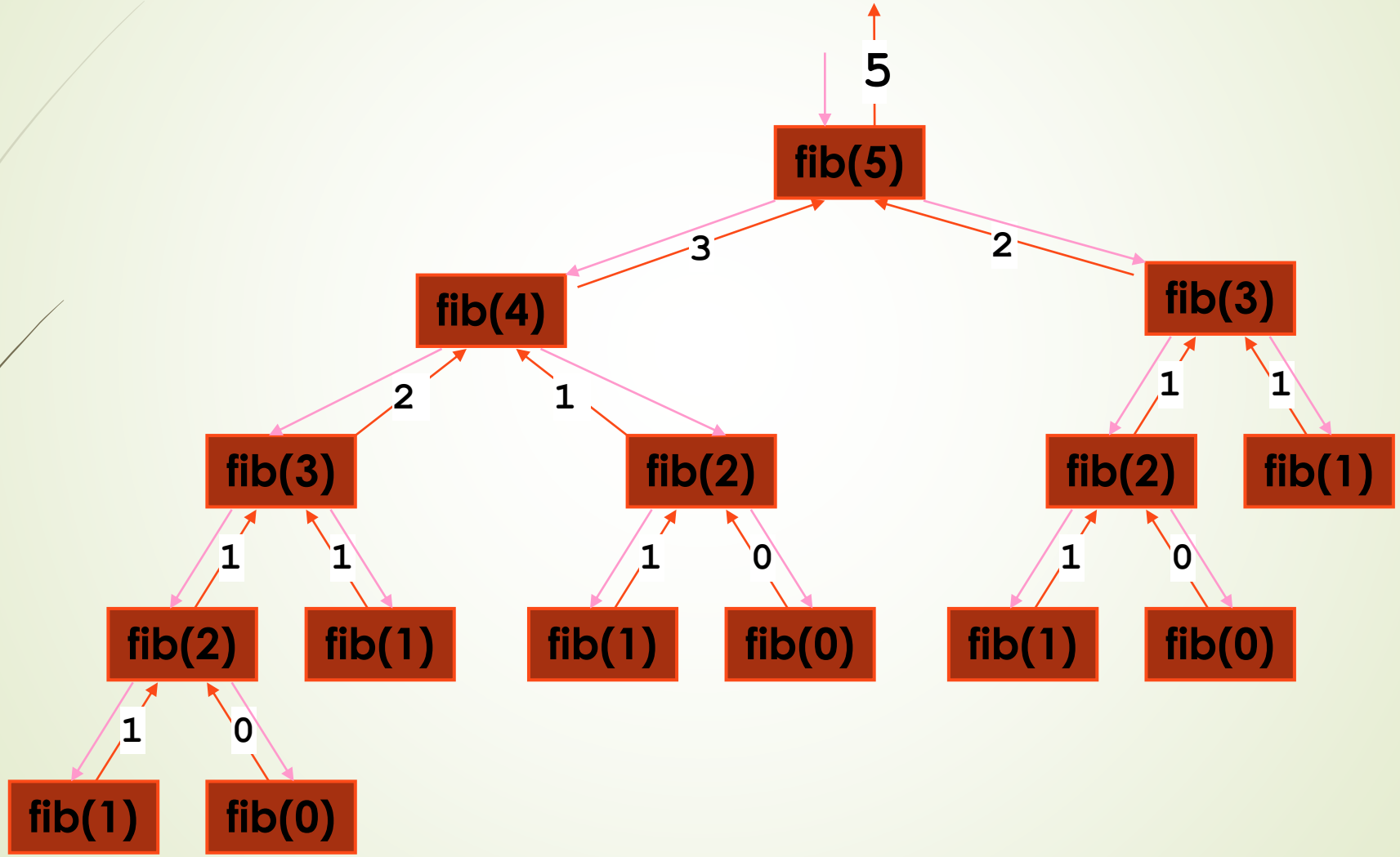
- Each term is sum of preceding two terms

$$\text{If } n = 0 \text{ or } n = 1, \text{ then } F_n = n.$$

$$\text{If } n > 1, \text{ then } F_n = F_{n-2} + F_{n-1}.$$

```
FIBONACCI(FIB, N)
1. If N = 0 or N = 1, then: Set FIB := N, and Return.
2. Call FIBONACCI(FIBA, N – 2).
3. Call FIBONACCI(FIBB, N – 1).
4. Set FIB := FIBA + FIBB.
5. Return.
```

# Function Analysis for call `fib(5)`



# Ackermann Function

**(a)** If $m = 0$, then $A(m, n) = n + 1$.

**(b)** If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$.

**(c)** If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$

# Simulating Recursion: Disadvantages

- Recursive algorithms have more overhead than similar iterative algorithms
  - Because of the repeated method calls (storing and removing data from call stack)
  - This may also cause a “stack overflow” when the call stack gets full
- Some recursive algorithms are inherently inefficient
  - An example of this is the recursive Fibonacci algorithm which repeats the same calculation again and again
  - Look at the number of times fib(2) is called
- Some languages (FORTRAN, COBOL) does not support recursion

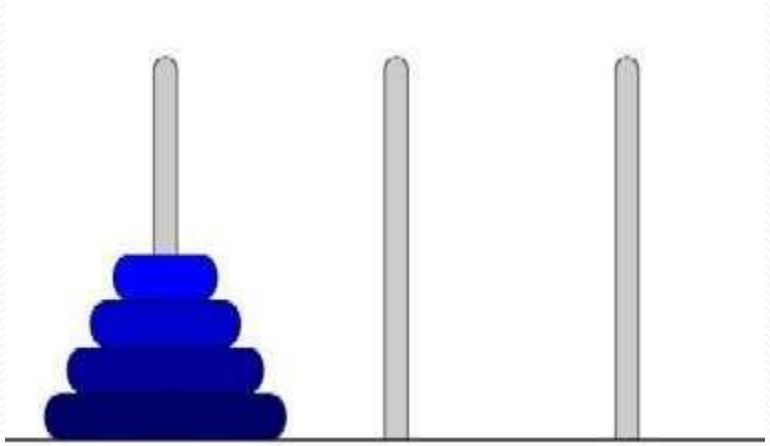
# Simulating Recursion

- It is often useful to derive a solution using recursion and implement it iteratively
  - Sometimes this can be quite challenging!

- To make recursive algorithm efficient:
  - Generic method: store all results in some data structure, and before making the recursive call, check whether the problem has been solved.
  - Make iterative version.

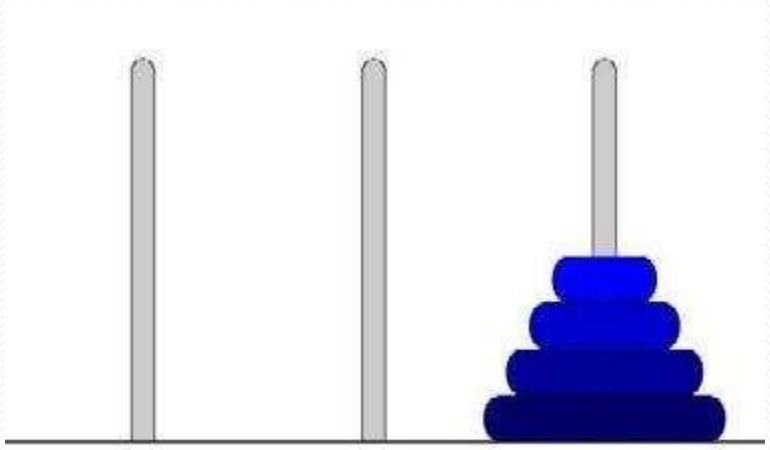
# Tower of Hanoi

- Tower of Hanoi is a mathematical puzzle invented by a French Mathematician in 1883.
- The game starts by having few discs stacked in increasing order of size. The number of discs can vary, but there are only three pegs.

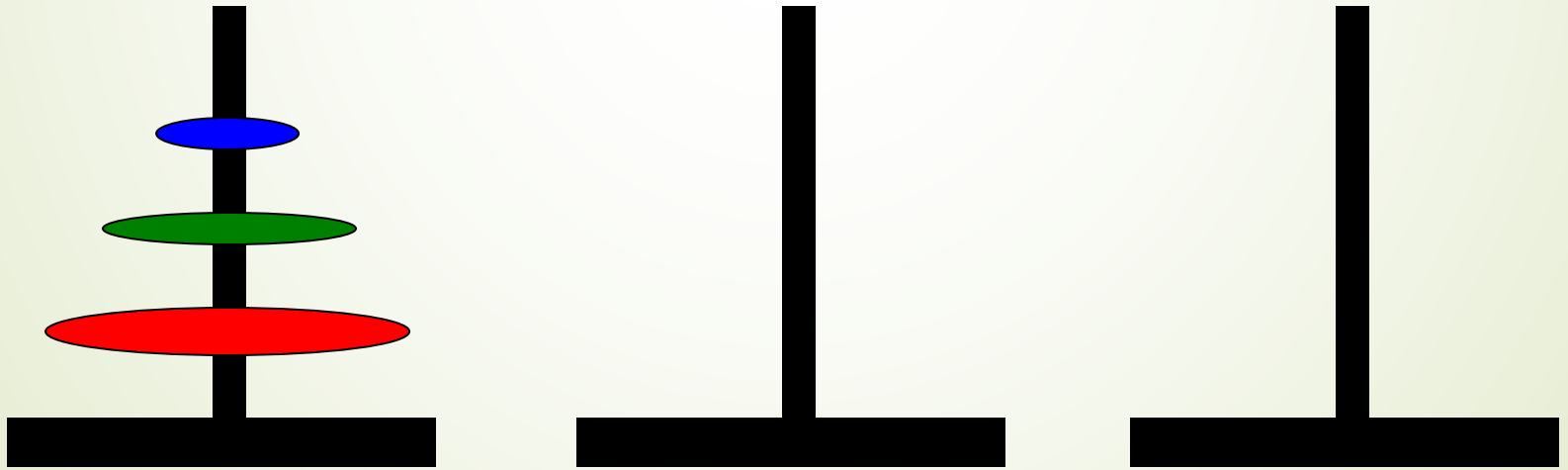


# Tower of Hanoi

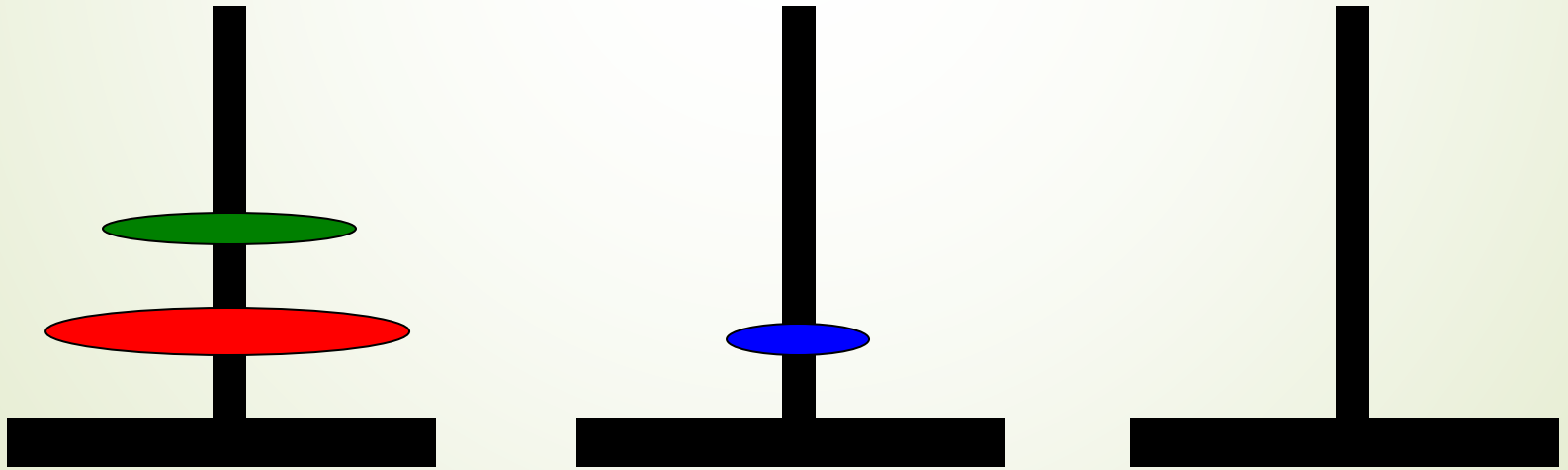
- The Objective is to transfer the entire tower to one of the other pegs. However you can only move one disk at a time and you can never stack a larger disk onto a smaller disk. Try to solve it in fewest possible moves.



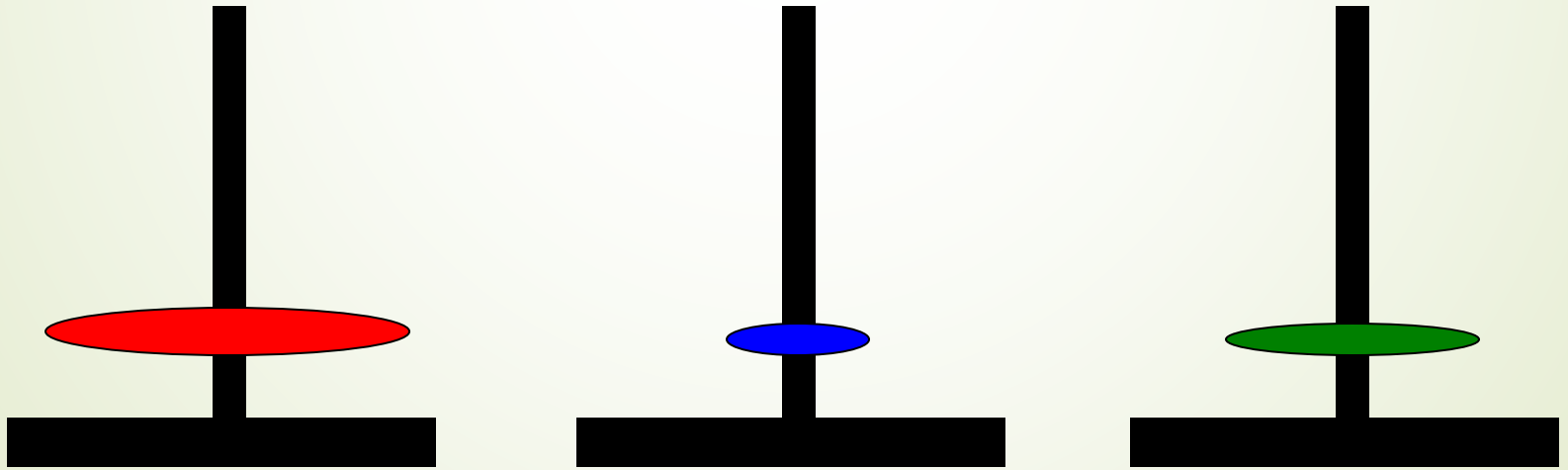
# Tower of Hanoi



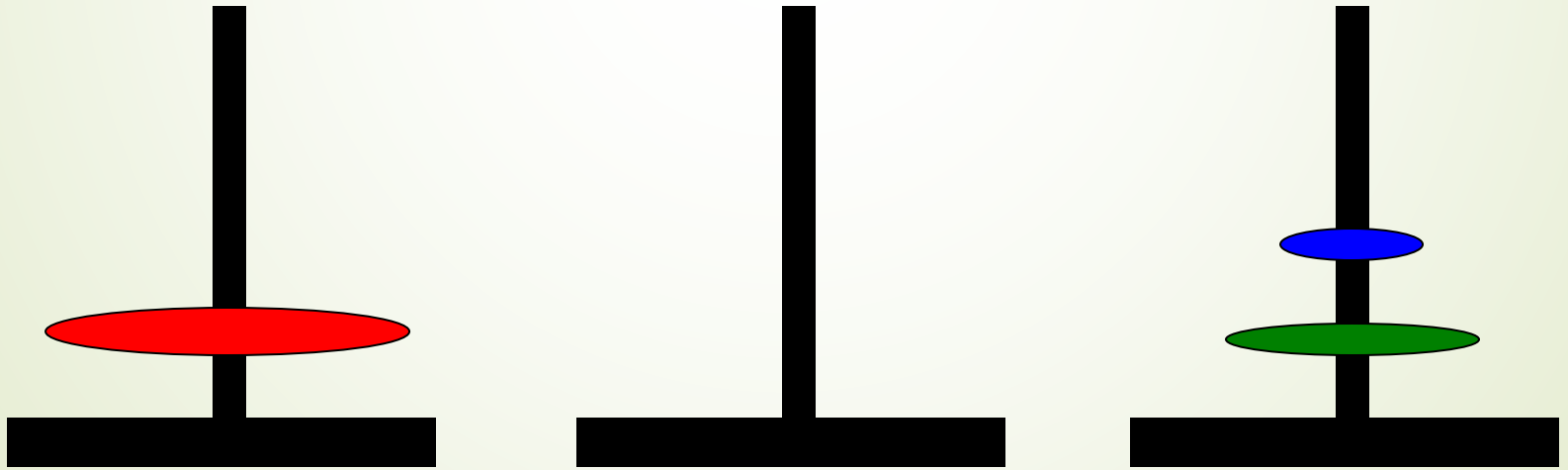
# Tower of Hanoi



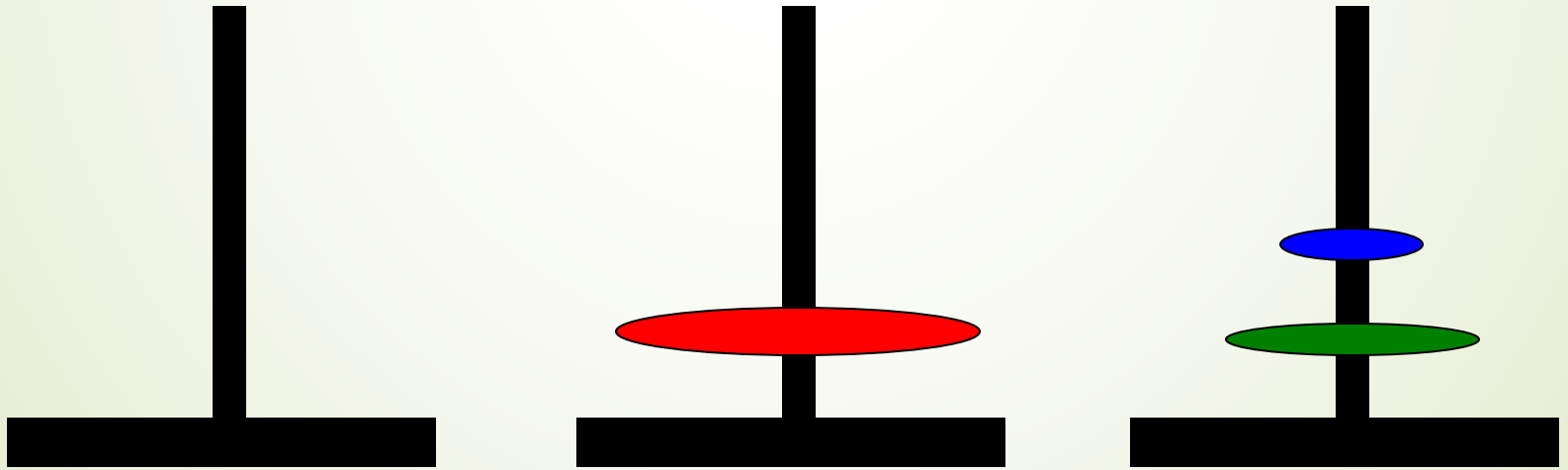
# Tower of Hanoi



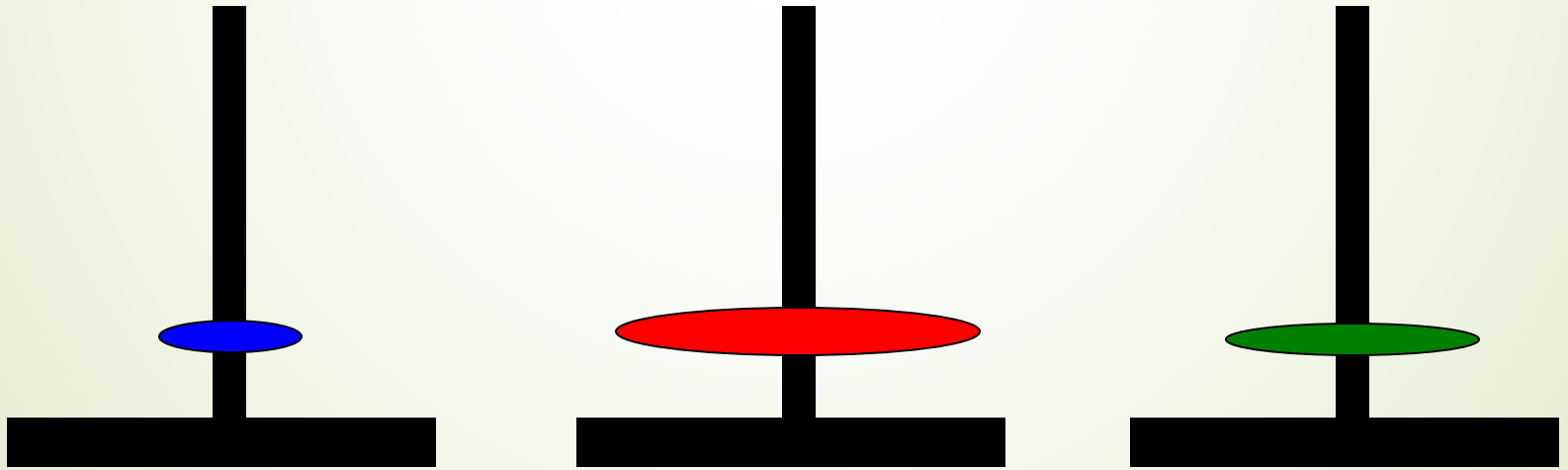
# Tower of Hanoi



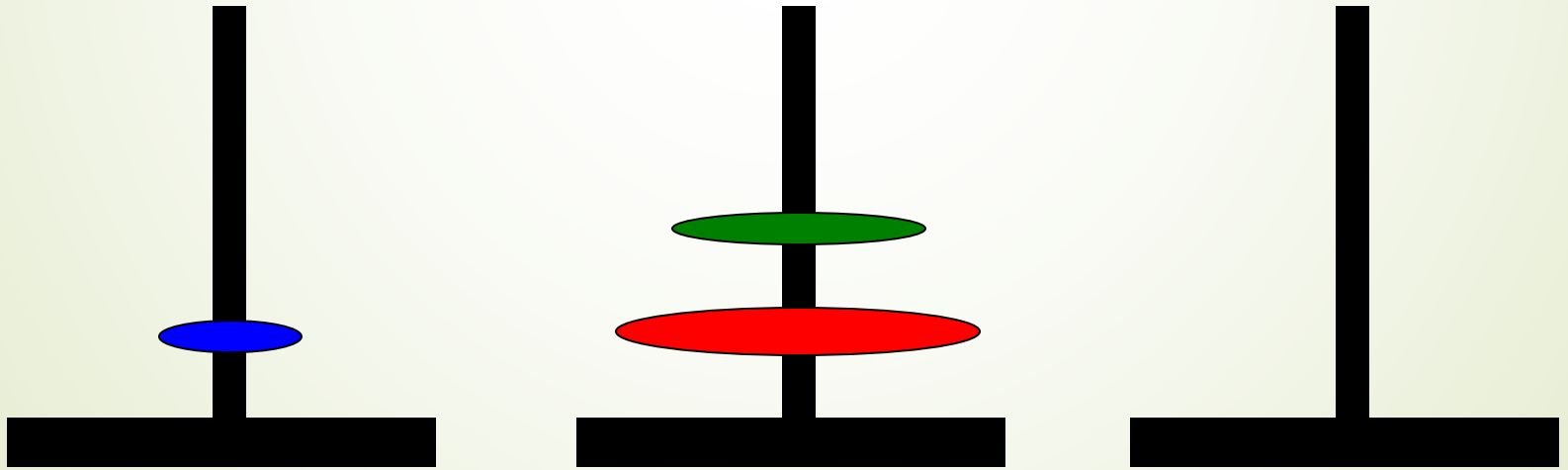
# Tower of Hanoi



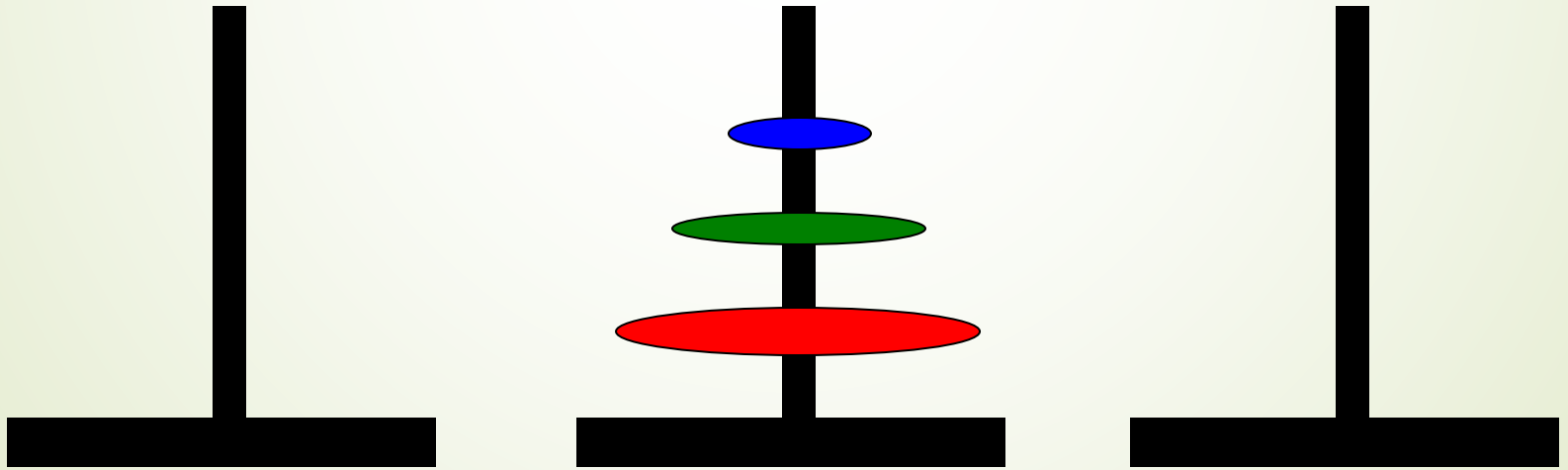
# Tower of Hanoi



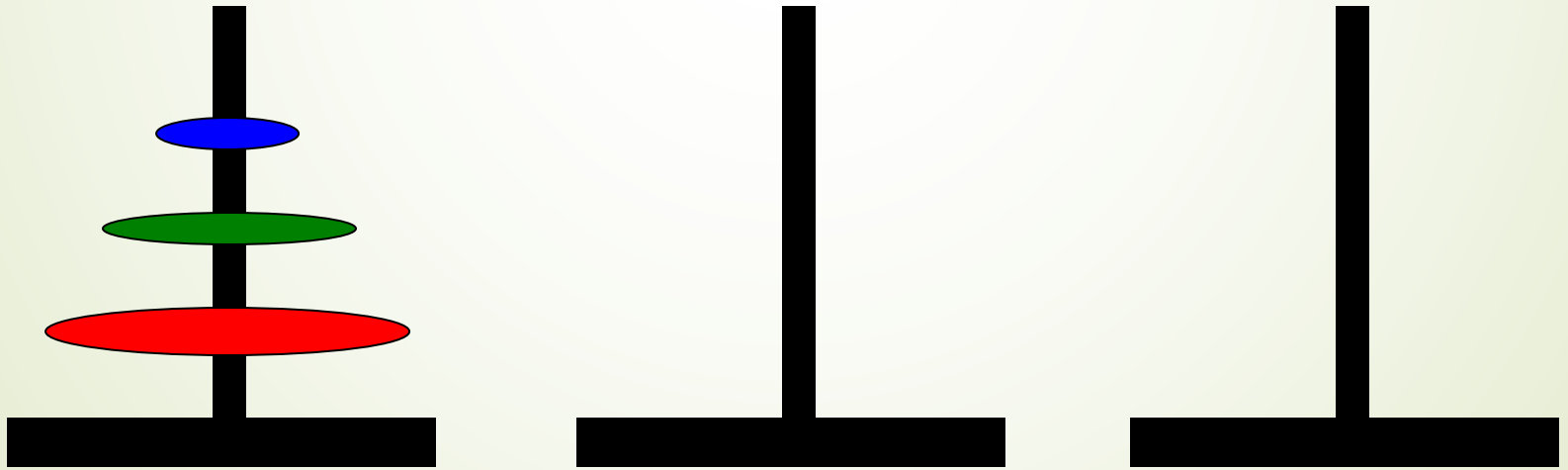
# Tower of Hanoi



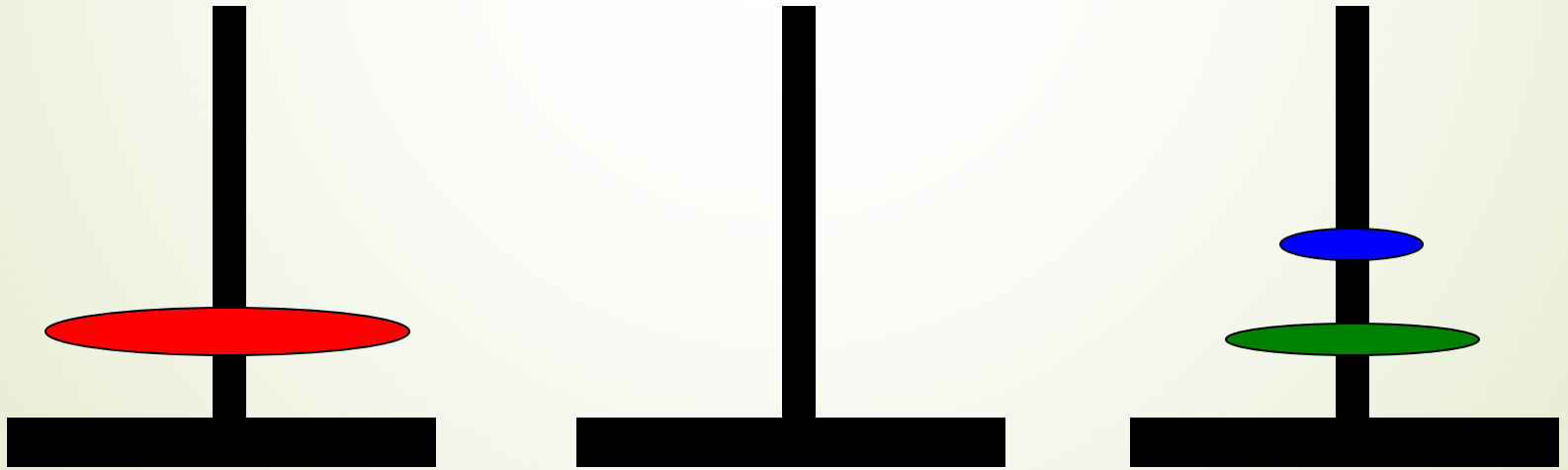
# Tower of Hanoi



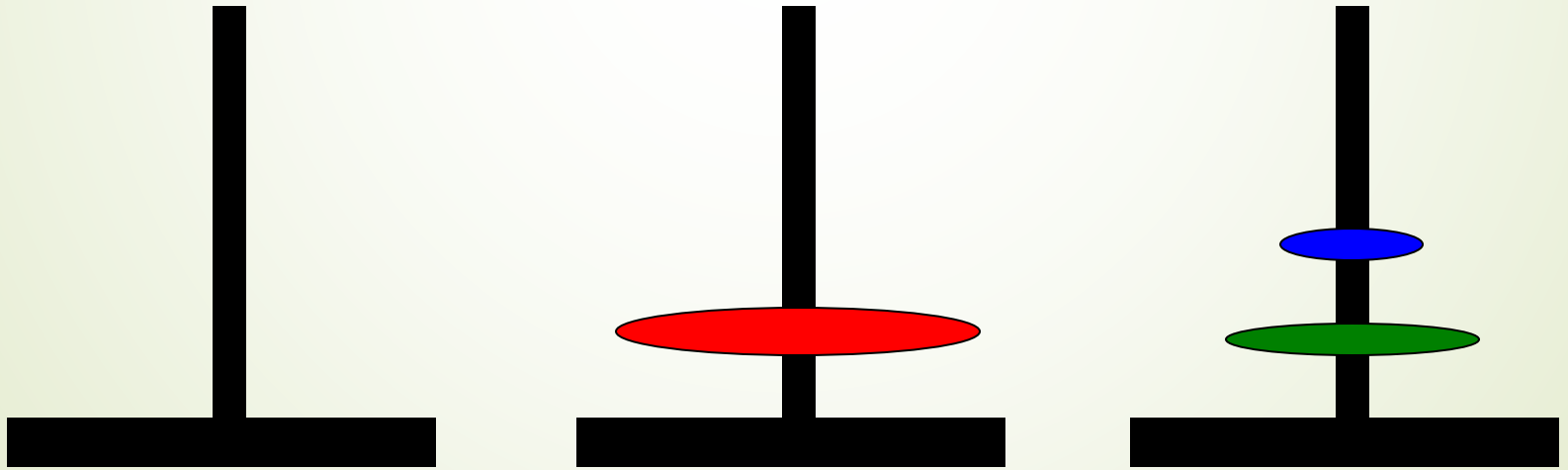
# Recursive Solution



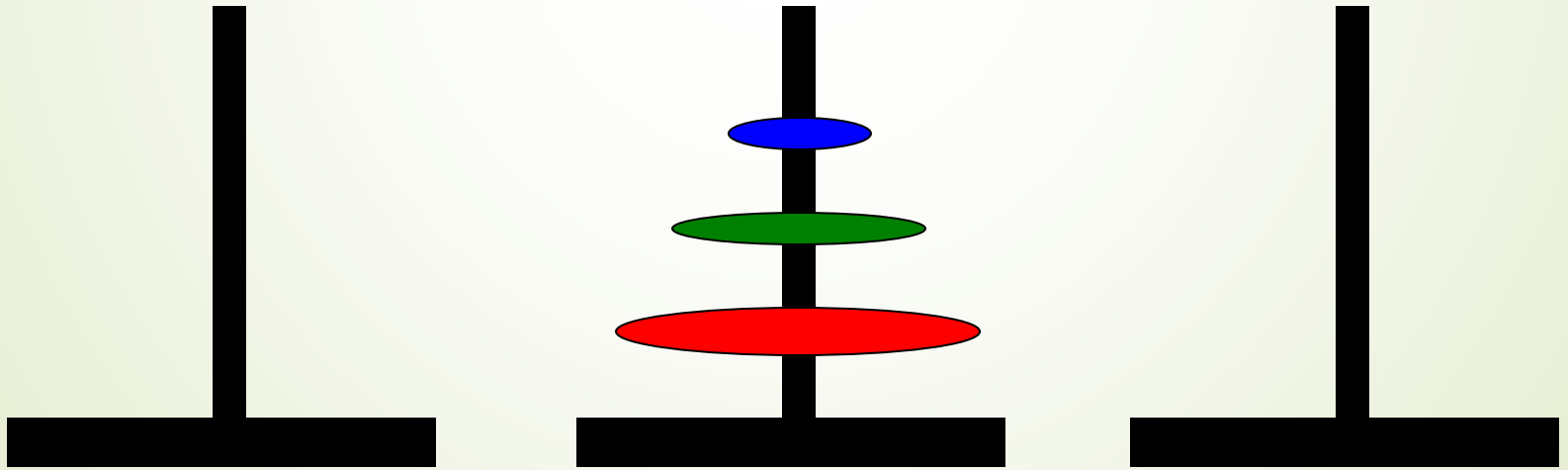
# Recursive Solution



# Recursive Solution

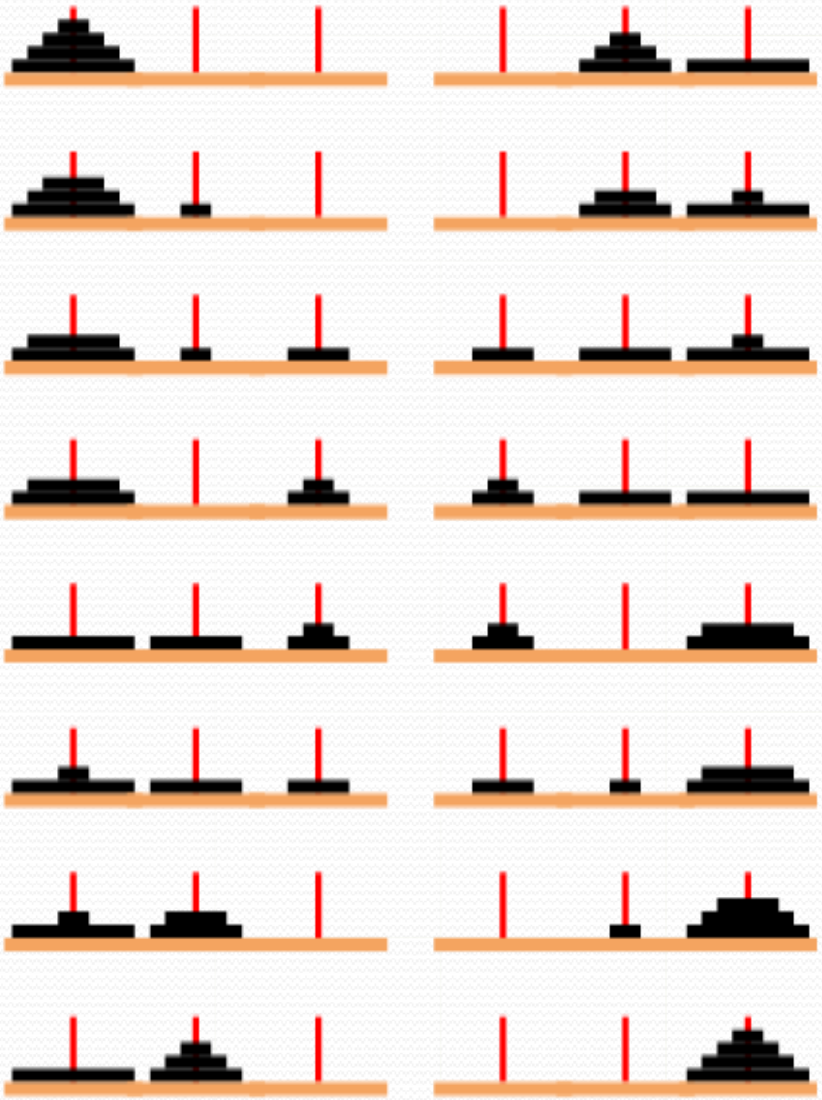


# Recursive Solution



# Tower of Hanoi

How to solve the 4discs



# Tower of Hanoi

- Recursive Solution for the Tower of Hanoi with algorithm.
- Let's call the three peg Src(Source), Aux(Auxiliary)and Dst(Destination).
  - Move the top N – 1 disks from the Source to Auxiliary tower.
  - Move the Nth disk from Source to Destination tower.
  - Move the N – 1 disks from Auxiliary tower to Destination tower. Transferring the top N – 1 disks from Source to Auxiliary tower can again be thought of as a fresh problem and can be solved in the same manner.

# Tower of Hanoi

TOWER(N, BEG, AUX, END)

1. If N = 1 then
   a. Write BEG --> END
   b. Return
2. [Move N-1 disks from peg BEG to peg AUX]
   Call TOWER(N-1, BEG, END, AUX)
3. Write BEG --> END
4. [Move N-1 disks from peg AUX to peg END]
   Call TOWER(N-1, AUX, BEG, END)
5. Return.

# Tower of Hanoi

For N = 4 we get the following sequence

1. Move from Src to Aux
2. Move from Src to Dst
3. Move from Aux to Dst
4. Move from Src to Aux
5. Move from Dst to Src
6. Move from Dst to Aux
7. Move from Src to Aux
8. Move from Src to Dst
9. Move from Aux to Dst
10. Move from Aux to Src
11. Move from Dst to Src
12. Move from Aux to Dst
13. Move from Src to Aux
14. Move from Src to Dst
15. Move from Aux to Dst

# Tower of Hanoi

- How many moves will it take to transfer n disks from the left post to the right post?

- for 1 disk it takes 1 move to transfer 1 disk from post A to post C;
- for 2 disks, it will take 3 moves: $2M + 1 = 2(1) + 1 = 3$
- for 3 disks, it will take 7 moves: $2M + 1 = 2(3) + 1 = 7$
- for 4 disks, it will take 15 moves: $2M + 1 = 2(7) + 1 = 15$
- for 5 disks, it will take 31 moves: $2M + 1 = 2(15) + 1 = 31$
- for 6 disks… ?

# Tower of Hanoi

- Explicit Pattern
- Number of Disks — Number of Moves

| Number of Disks | Number of Moves |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |
| 5 | 31 |

- Powers of two help reveal the pattern:
- Number of Disks (n) — Number of Moves

| Number of Disks (n) | Number of Moves |
|---|---|
| 1 | 2^1 - 1 = 2 - 1 = 1 |
| 2 | 2^2 - 1 = 4 - 1 = 3 |
| 3 | 2^3 - 1 = 8 - 1 = 7 |
| 4 | 2^4 - 1 = 16 - 1 = 15 |
| 5 | 2^5 - 1 = 32 - 1 = 31 |

# Simulating Recursion: Translating recursive procedure into non-recursive procedure

**(1)** A stack STPAR for each parameter PAR
**(2)** A stack STVAR for each local variable VAR
**(3)** A local variable ADD and a stack STADD to hold return addresses

# Simulating Recursion: Translating recursive procedure into non-recursive procedure

**(1)** Preparation.

**(a)** Define a stack STPAR for each parameter PAR, a stack STVAR for each local variable VAR, and a local variable ADD and a stack STADD to hold return addresses.

**(b)** Set TOP := NULL.

**(2)** Translation of "Step K. Call P."

**(a)** Push the current values of the parameters and local variables onto the appropriate stacks, and push the new return address [Step] K + 1 onto STADD.

**(b)** Reset the parameters using the new argument values.

**(c)** Go to Step 1. [The beginning of the procedure P.]

**(3)** Translation of "Step J. Return."

**(a)** If STADD is empty, then: Return. [Control is returned to the main program.]

**(b)** Restore the top values of the stacks. That is, set the parameters and local variables equal to the top values on the stacks, and set ADD equal to the top value on the stack STADD.

**(c)** Go to Step ADD.

# Non-recursive Tower of Hanoi

TOWER(N, BEG, AUX, END)

**0**. Set TOP := NULL.

**1**. If N = l, then:

**(a)** Write: BEG → END.

**(b)** Go to Step 5.

[End of If structure.]

**2**. [Translation of "Call TOWER(N – 1, BEG, END, AUX)."]

**(a)** [Push current values and new return address onto stacks.]

**(i)** Set TOP := TOP + 1.

**(ii)** Set STN[TOP] := N, STBEG[TOP] := BEG,

STAUX[TOP] := AUX, STEND[TOP] := END,
STADD[TOP] := 3.

**(b)** [Reset parameters.]

Get N := N – 1, BEG := BEG, AUX := END, END := AUX.

**(c)** Go to Step 1.

**3**. Write: BEG → END.

**4**. [Translation of "Call TOWER(N – 1, AUX, BEG, END)."]

**(a)** [Push current values and new return address onto stacks.]

**(i)** Set TOP := TOP + 1.

**(ii)** Set STN[TOP] := N, STBEG[TOP] := BEG,
STAUX[TOP] := AUX, STEND[TOP] := END,
STADD[TOP] := 5.

**(b)** [Reset parameters.]

Set N := N – 1, BEG := AUX, AUX := BEG, END := END.

**(c)** Go to Step 1.

**5**. [Translation of "Return."]

**(a)** If TOP := NULL, then: Return.

**(b)** [Restore top values on stacks.]

**(i)** Set N := STN[TOP], BEG := STBEG[TOP],
AUX := STAUX[TOP], STEND[TOP],
ADD := STADD[TOP].

**(ii)** Set TOP := TOP – 1.

**(c)** Go to Step ADD.