Applications of Stack

- Polish Notations
- Evaluation of Postfix Expression

Different forms of arithmetic Expressions

- Infix :- the general mathematics notation, where the operator is written in-between the operands.
  - For E.g. :- A + B
- Prefix :- operator is place before the operands, also called as **Polish notation**.
  - E.g. :- + A B
- Postfix :- operator is place after the operands, also called as **Reverse Polish notation or simply Suffix Notation**.
  - E.g. :- A B +

Need for polish and suffix notation

- Computer doesn't understand the regular mathematical expression. So, complier needs to converts the regular expression into polish (prefix) notation or postfix(reverse polish) expression for performing evaluation operation.

Need for polish and suffix notation

Infix Notation	Prefix Notation	Postfix Notation
A + B	+ AB	AB +
(A - C) + B	+ - ACB	AC – B +
A + (B * C)	+ A * BC	ABC *+
(A+B)/(C-D)	/+ AB – CD	AB + CD -/
(A + (B * C))/(C – (D * B))	/+ A * BC – C * DB	ABC * + CDB * - /

operator precedence rules

Operators	Symbols
Parenthesis	( ), {}, [ ]
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+ , -

Associativity rule

- The first preference is given to the parenthesis
- Next preference is given to the exponents.
- In the case of multiple exponent operators, then the operation will be applied from right to left.

For example:

- 2^2^3 = 2 ^ 8
- = 256.
- After exponent, multiplication, and division operators are evaluated. If both the operators are present in the expression, then the operation will be applied from left to right.
- The next preference is given to addition and subtraction. If both the operators are available in the expression, then we go from left to right.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Infix to Postfix Expression

//Input: Infix Expression A

//Output: Equivalent Postfix Expression B

Step 1. Scan A from left to right and repeat step 3 to 6 for each element, x, of A until the stack is empty

Step 2. If x is an operand, add it to B

Step 3. If x is a left parenthesis, '(', push it onto the stack

Step 4. If x is a right parenthesis, ')', then

1. Repeatedly pop from the STACK and add to B each operator (on the top of STACK) until a left parenthesis, '(', is encountered
2. Remove the left parenthesis, '('. (Do not add left parenthesis to B)

Step 5. If x is an operator, then

1. Repeatedly pop from the STACK and add to B each operator (on the top of stack) which has the same precedence as or higher precedence than x
2. Add x to STACK

Step 6. Pop the remaining operators from stack and add it to B

Step 7. Exit

Example: A * B + C

A * B + C becomes A B * C +

current symbol	Opstack	Poststack
A		A
*	*	A
B	*	AB
+	+	AB* {pop and print the '*' before pushing the '+'}
C	+	AB*C
		AB*C+

Example: A * (B + C)

A * (B + C) becomes **A B C + ***

current symbol	Opstack	Poststack
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
		ABC+*

Example: A * (B + C)

A * (B + C) becomes **A B C + ***

current symbol	Opstack	Poststack
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
		ABC+*

Example: (A + (B * C – (D / E ^ F) * G) * H)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
(	A	+	(	B	*	C	-	(	D	/	E	^	F	)	*	G	)	*	H	)

	10
	9
	8
	7
	6
	5
	4
	3
	2
	1
	0

Operator Stack

Example: (A + (B * C – (D / E ^ F) * G) * H)

Symbol Scanned		Stack	Expression Y
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 *
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 ↑ /
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 * +

Evaluation of Postfix Expression

//Input: Postfix Expression, A

//Output: Result of the evaluation of A

Scan the expression from left to right

Step 1. Read the next element, x, in A # first element for first time #

Step 2. If x is an operand then

i. Push the element in the stack

Step 3. If x is an operator then

i. Pop two operands from the stack i.e., a = pop() and b = pop()
   # POP one operand in case of unary operator #
ii. Evaluate the expression formed by the two operands and the operator, i.e., *b x a*
iii. Push the result of the expression into the stack.

Step 4. If no more elements then

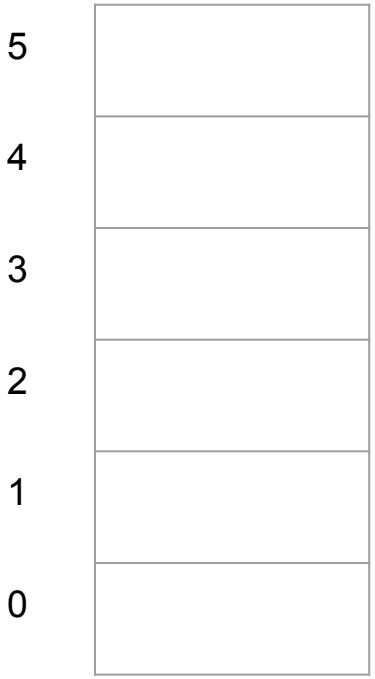
POP the result

else

goto step 1

Step 5. Exit

Example: **4 2 - 3 5 1 + * +**



Operand Stack

Evaluation of Postfix Expression:
6 3 – 2 *

Recursion

- **power(2,3):** 2^3
  - Use recursive definition: $\mathbf{x^n = x^{n-1} * x}$
  - Translating recursive call: **power(x, n-1) * x**

```
def Power(x,n):
    if n==0:
        return 1
    return (Power(x,n-1) * x)
```

Power(2, 3) results in more recursive calls:

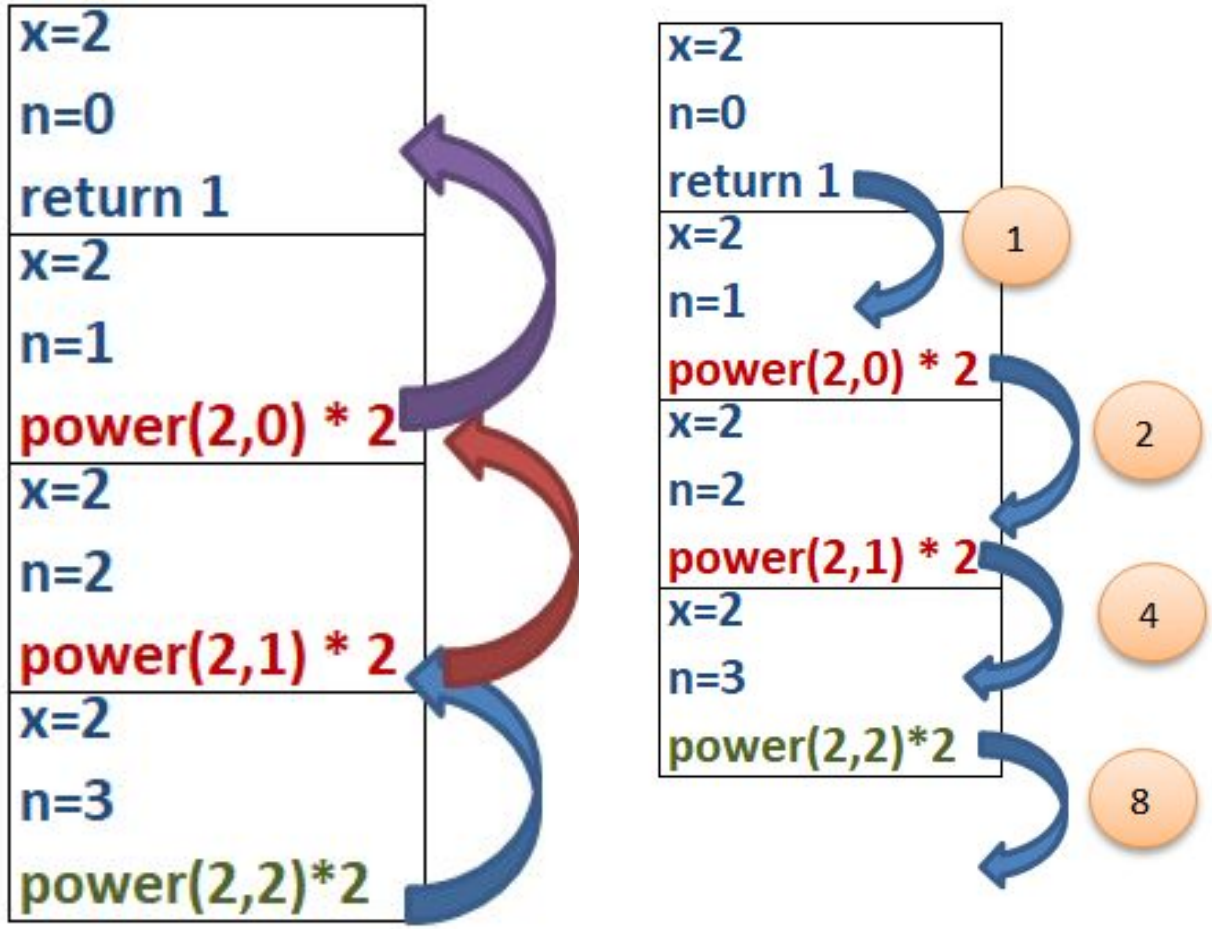
power(2, 3) is power(2, 2) * 2
Power(2, 2) is power(2, 1) * 2
Power(2, 1) is power(2, 0) * 2
Power (2, 0) is 1 (stopping case)

Recursion

- **power(2,3):** 2^3
  - Use recursive definition: $x^n = x^{n-1} * x$
  - Translating recursive call: **power(x, n-1) * x**

```
def Power(x,n):
    if n==0:
    return 1
    return (Power(x,n-1) * x)
```

Power(2, 3) results in more recursive calls:
power(2, 3) is power(2, 2) * 2
Power(2, 2) is power(2, 1) * 2
Power(2, 1) is power(2, 0) * 2
Power (2, 0) is 1 (stopping case)



factorial of a number

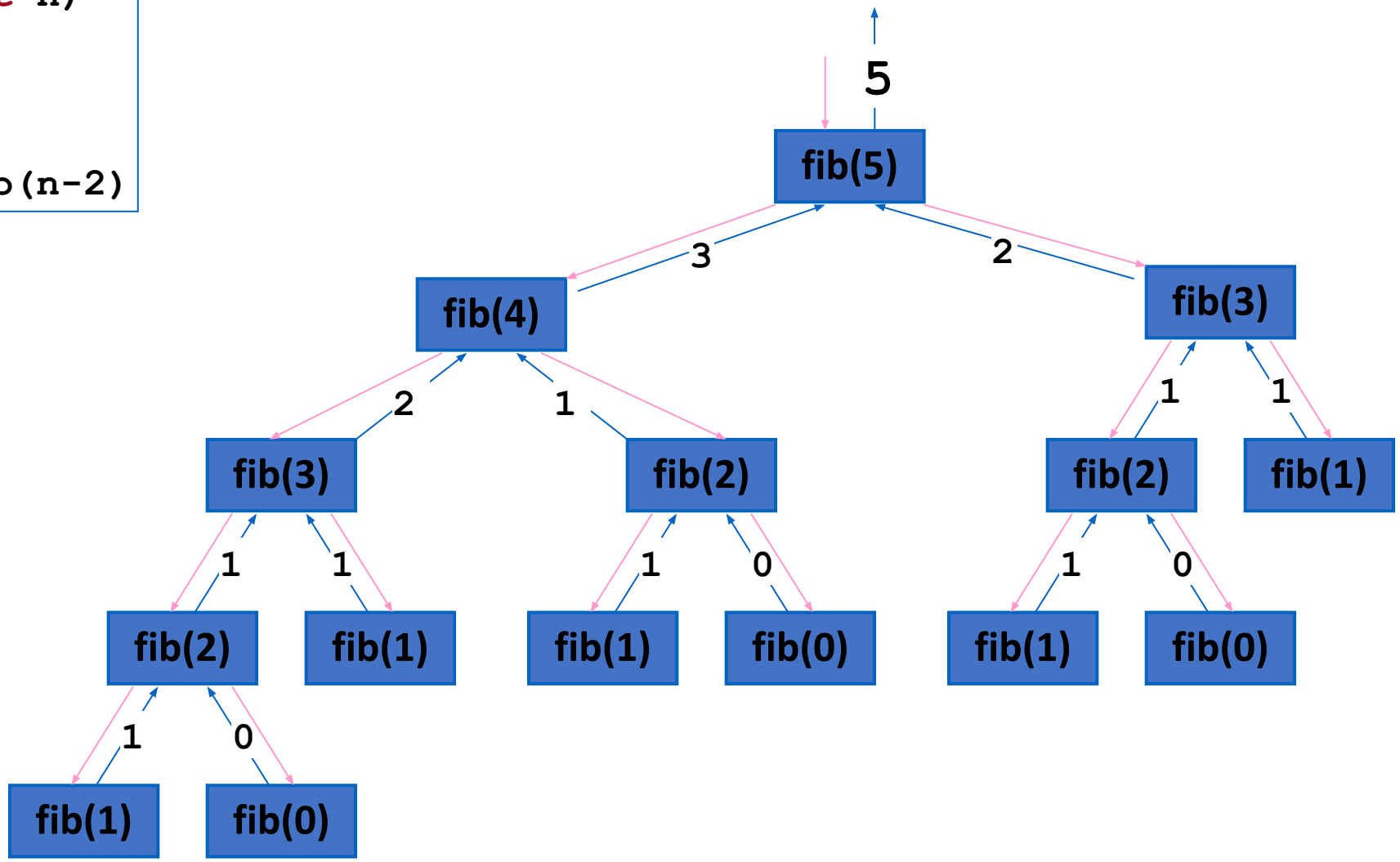
```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

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

Recursion Example = nth Fibonacci Number

```
public static int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```

Function Analysis for call `fib(5)`



Recursion Example – Towers of Hanoi

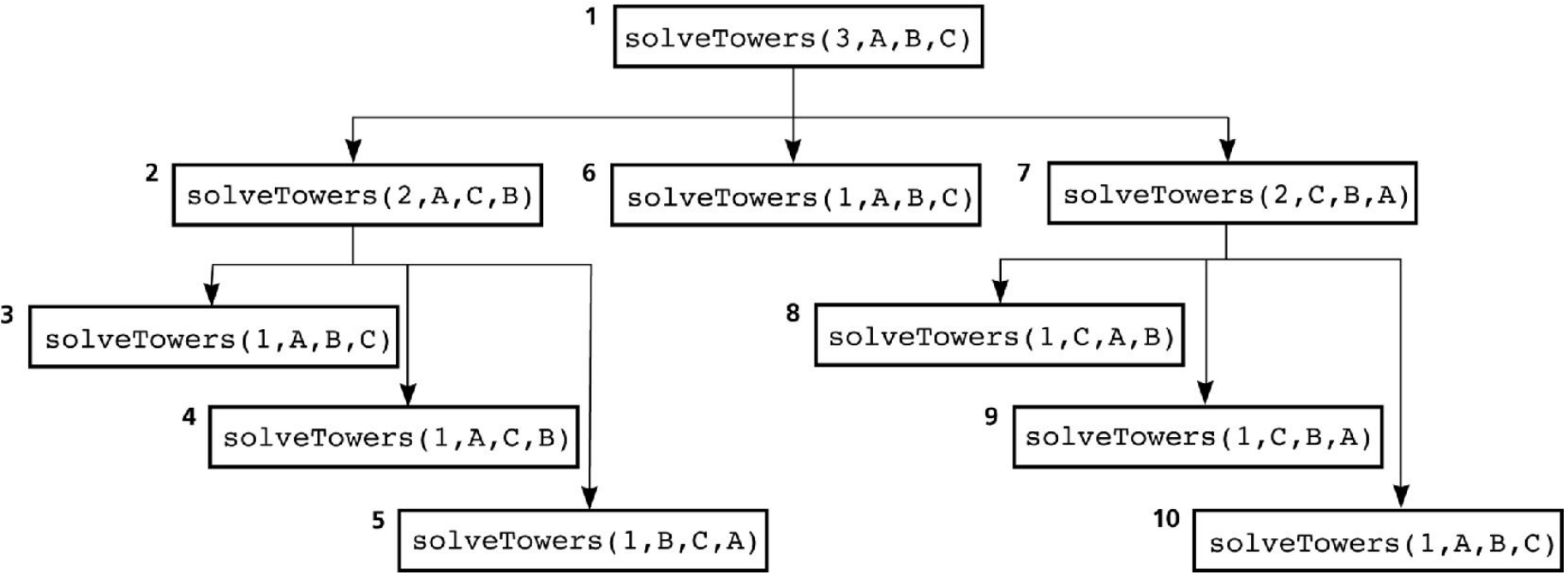
- https://www.mathplayground.com/logic_tower_of_hanoi.html

Hanoi towers

```
TowersOfHanoi(n,source,destination,spare)
  if (n == 1):
    print("Move top disk from pole " + source + " to pole " + destination)
  else:
    TowersOfHanoi(n-1, source, spare, destination) #X
    print("Move top disk from pole " + source + " to pole " + destination)#Y
    TowersOfHanoi(n-1, spare, destination, source)#Z
```

Recursion tree:

The order of recursive calls that results from ***solveTowers(3,A,B,C)***

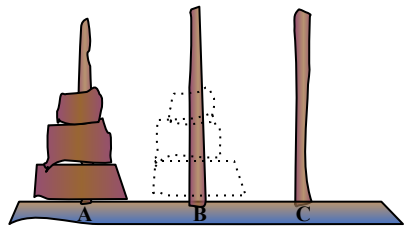



```
TowersOfHanoi(n,source,destination,spare)
    if (n == 1):
      print("Move top disk from pole " + source + " to pole " + destination)
    else:
      TowersOfHanoi(n-1, source, spare, destination)
      print("Move top disk from pole " + source + " to pole " + destination)
      TowersOfHanoi(n-1, spare, destination, source)
```

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

The initial call 1 is made, and **solveTowers** begins execution:

```
count  = 3
source = A
dest   = B
spare  = C
```

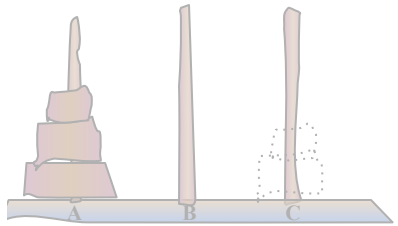


At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count  = 3
source = A
dest   = B
spare  = C
```

X

```
count  = 2
source = A
dest   = C
spare  = B
```



At point X, recursive call 3 is made, and the new invocation of the method begins execution:

```
count  = 3
source = A
dest   = B
spare  = C
```

X

```
count  = 2
source = A
dest   = C
spare  = B
```

X

```
count  = 1
source = A
dest   = B
spare  = C
```


This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count  = 3
source = A
dest   = B
spare  = C
```

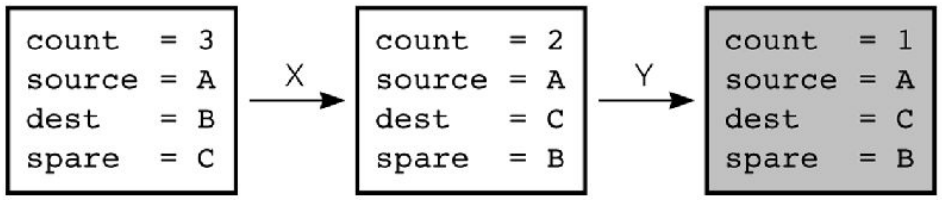
X

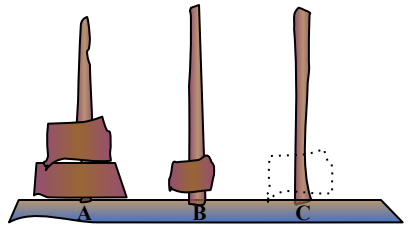
```
count  = 2
source = A
dest   = C
spare  = B
```

```
count  = 1
source = A
dest   = B
spare  = C
```

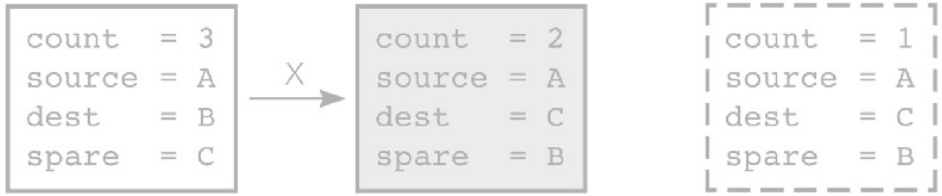
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

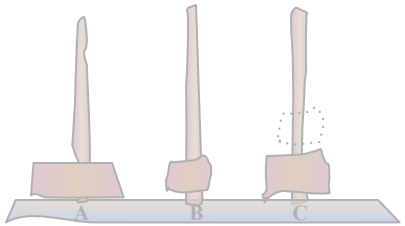
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:





This is the base case, so a disk is moved, the return is made, and the method continues execution.





At point Z, recursive call 5 is made, and the new invocation of the method begins execution:

This is the base case, so a disk is moved, the return is made, and the method continues execution.

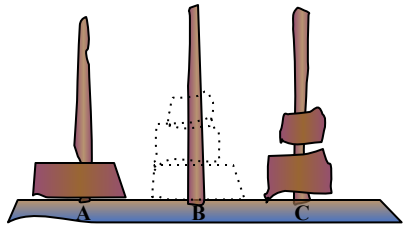
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

This invocation completes, the return is made, and the method continues execution.

```
count  = 3
source = A
dest   = B
spare  = C
```

```
count  = 2
source = A
dest   = C
spare  = B
```

```
count  = 1
source = B
dest   = C
spare  = A
```

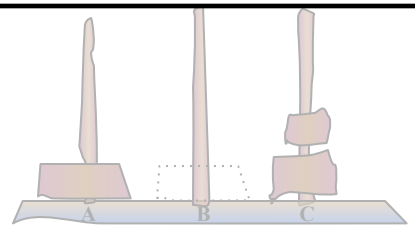


At point Y, recursive call 6 is made, and the new invocation of the method begins execution:

```
count  = 3
source = A
dest   = B
spare  = C
```

Y

```
count  = 1
source = A
dest   = B
spare  = C
```



This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count  = 3
source = A
dest   = B
spare  = C
```

```
count  = 1
source = A
dest   = B
spare  = C
```


At point Z, recursive call 7 is made, and the new invocation of the method begins execution:

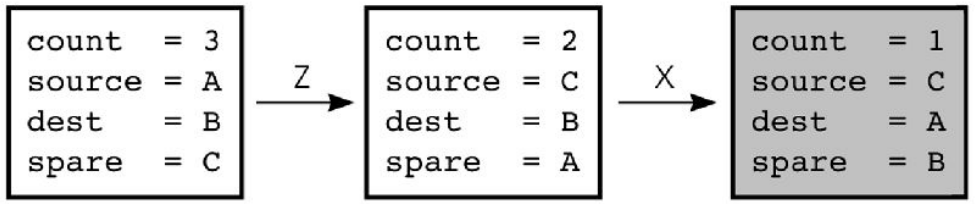
```
count  = 3
source = A
dest   = B
spare  = C
```

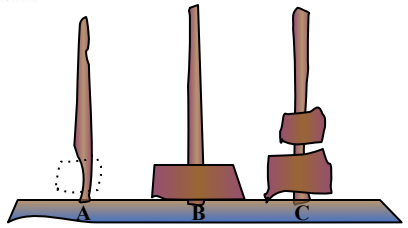
Z

```
count  = 2
source = C
dest   = B
spare  = A
```

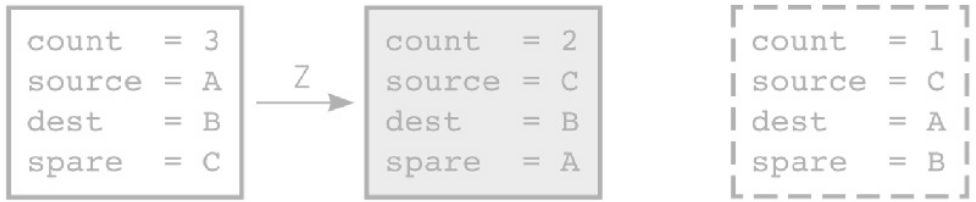
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

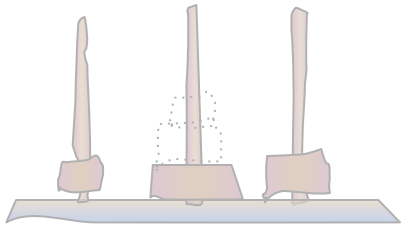
At point X, recursive call 8 is made, and the new invocation of the method begins execution:





This is the base case, so a disk is moved, the return is made, and the method continues execution.



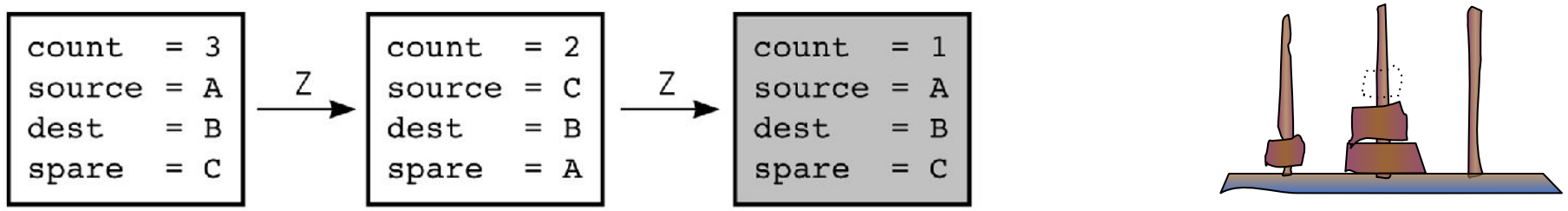


At point Y, recursive call 9 is made, and the new invocation of the method begins execution:

This is the base case, so a disk is moved, the return is made, and the method continues execution.

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

This invocation completes, the return is made, and the method continues execution.

Cost of Hanoi Towers

- How many moves is necessary to solve Hanoi Towers problem for N disks?
- moves(1) = 1
- moves(N) = moves(N-1) + moves(1) + moves(N-1)
- i.e.
  moves(N) = 2*moves(N-1) + 1
  $= 2^N - 1$