Subject: Data Structure Using C++ (CSET243)

BENNETT UNIVERSITY
THE TIMES GROUP

Week 3 Lecture

Structure, Pointer, DMA, Recursion, Binary Search, Tower of Hanoi

What, Why and How?

by

Dr Gaurav Kumar

Asst. Prof, Bennett University

Quick Recap of Last Week's Learnings

- Asymptotic Notations
- Algorithm & Time Complexity Analysis
- Analysis of Various Operations of 1D and 2D Arrays
- Visualized the Linear Searching Algorithm

Assessment Time

1. What is an algorithm?

a) A specific software program

b) A set of instructions designed to perform a specific task

c) A programming language

d) A type of computer hardware

Answer: b) A set of instructions designed to perform a specific task

Assessment Time

2. Which of the following is NOT a characteristic of a good algorithm?

a) Finite steps

b) Infinite loops

c) Well-defined inputs

d) Produces a correct output

Answer: b) Infinite loops

Assessment Time

3. What will be the output of the following code?

```
int a = 5;

cout << a++ << " " << ++a;
```

a) 5 6

b) 5 7

c) 6 7

d) 6 6

Answer: b) 5 7

Quick Recap of Visualization of Searching Algorithms

01 **Linear Search**

---

02 **Binary Search**

Visualization of Linear Search

8	3	1	2	4	5	6	7

Key Element = 4

Total Number of Comparison = 5

Key Element = 7

Total Number of Comparison = 8

Algorithm of linear search

Step 1- Start from the leftmost element of arr[] and one by one compare x with each element of arr[].

Step 2- If x matches with an element, return the index.

Step 3- If x doesn't match with any of the elements, return -1.

Visualization of Linear Search

8	3	1	2	4	5	6	7

Time Complexity Analysis

Searching the key element present at the first index of a list

Key Element = 8

Total Number of Comparison = 1

f(n) = 1 = Constant = O(1)

Best Case Time Complexity

Visualization of Linear Search

8	3	1	2	4	5	6	7

Time Complexity Analysis

Searching the key element present at the last index of a list

Key Element = 7

Total Number of Comparison = 7

for n elements $f(n) = n$

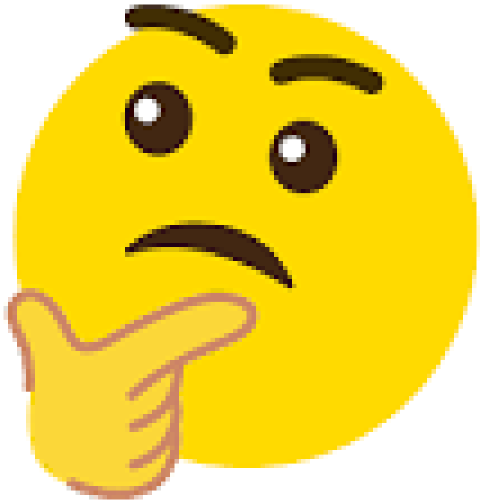
Worst Case Time Complexity

$f(n) = O(n)$

Visualization of Linear Search

8	3	1	2	4	5	6	7

Time Complexity Analysis



Average Case Time Complexity

Searching the key element present at any random index of a list

Key Element = 8 or 3 or 1 or 2 or 4 or 5…

Total Avg Time = All possible case time divided by number of cases

for n elements Average Time= (1+2+3+...n)/n = n(n+1)/2n = n+1/2

Avg Case Time Complexity

f(n) = O(n)

Quick Recap of Visualization of Searching Algorithm

2. Binary Search

- Binary search is a fast way to find an item in a **sorted list**.
- It follows the **divide and conquer approach** in which the list is divided into two halves, and the item is compared with the middle element of the list.
- If the match is found then, the location of the middle element is returned. Otherwise, we **search into either of the halves** depending upon the result produced through the match.

Visualization of Searching Algorithms

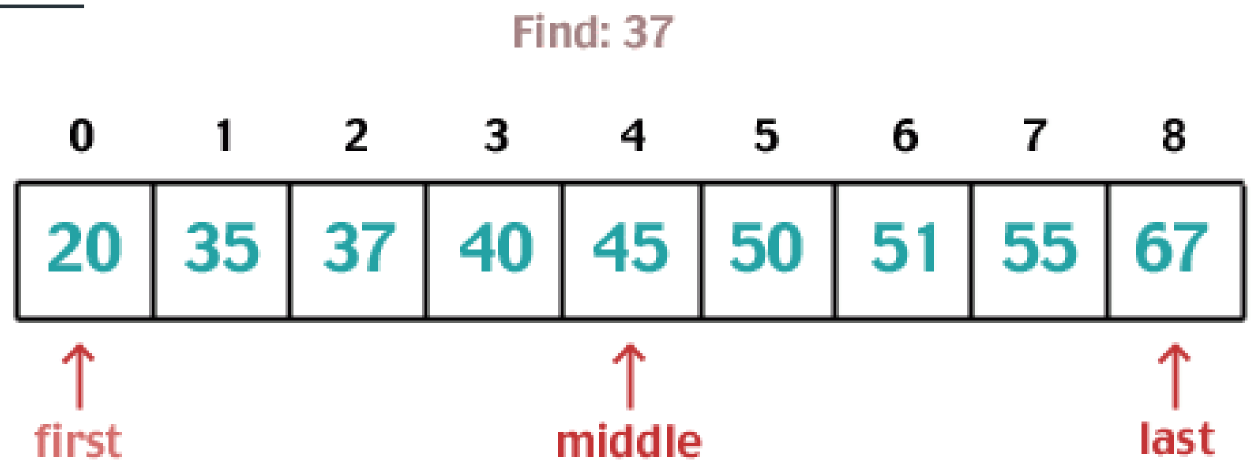
2. Binary Search

Search for 47

0	4	7	10	14	23	45	47	53

Visualization of Binary Searching Algorithms

1. Divide the search space into two halves by finding the middle index “mid”.
2. Compare the middle element of the search space with the key.



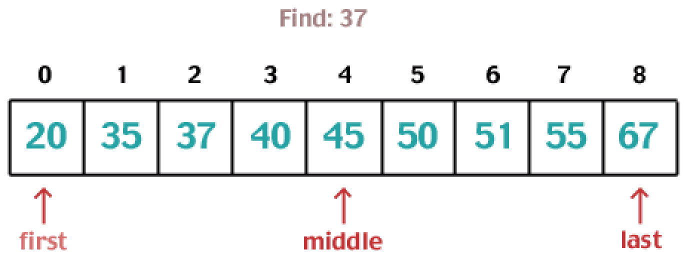
Find: 37

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
    1. If the key is smaller than the middle element, then the left side is used for next search.
    2. If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

Time Complexity of Binary Searching Algorithms

(Iterative Method Approach)



1. Divide the search space into two halves by finding the middle index “mid”.
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
    1. If the key is smaller than the middle element, then the left side is used for next search.
    2. If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

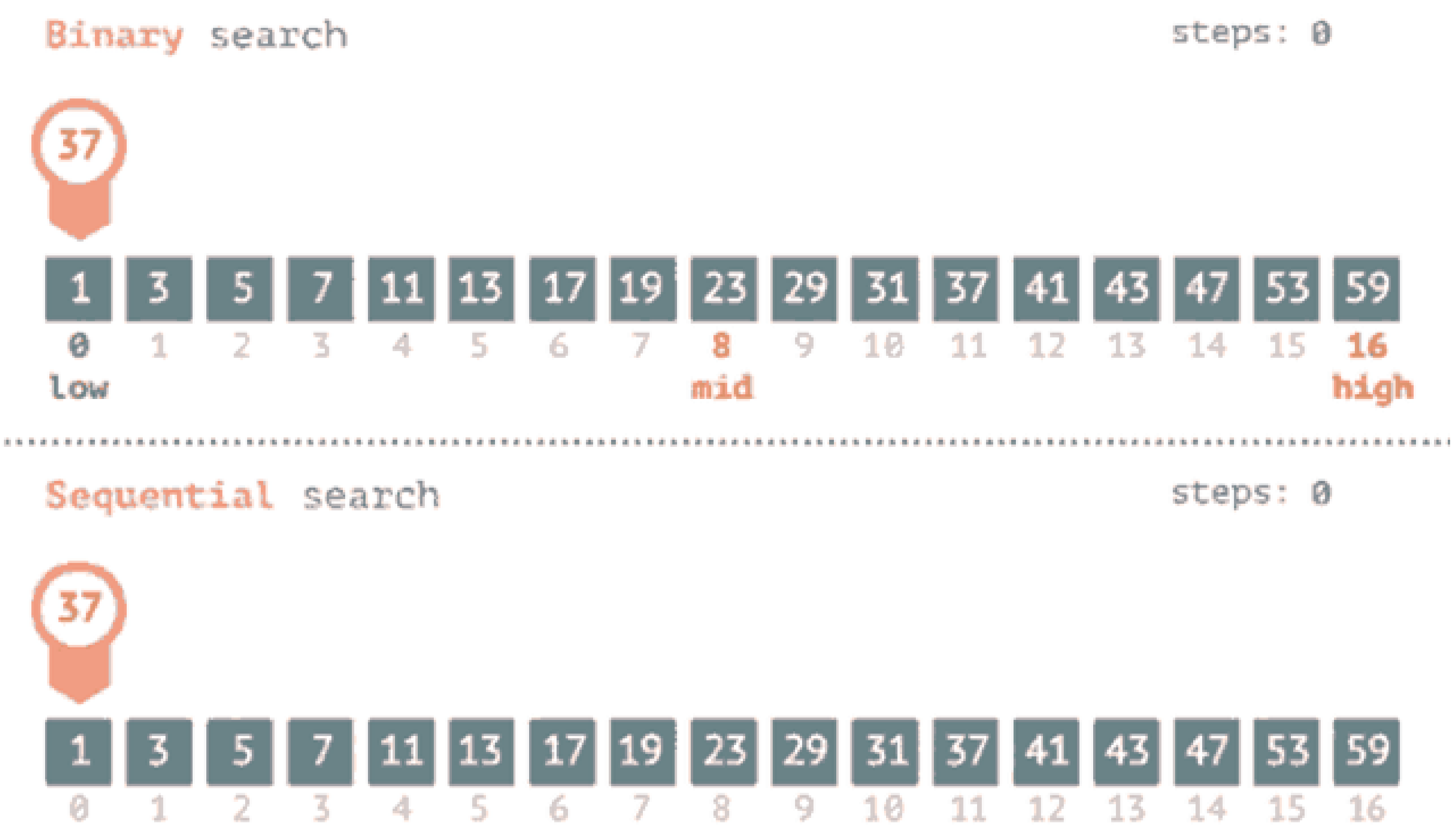
```
int binarySearch(int[] A, int x)
{
    int low = 0, high = A.length - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (x == A[mid])
        {
            return mid;
        }
        else if (x < A[mid])
        {
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }
    }
    return -1;
}
```

TC= O(logn)

??

Visualization of Time Complexity of Binary Searching Algorithm

TC= O(logn)



Number of Elements	Number of Times Mid Comparison	(Superscript)
2	1	---------→ 2^1
4	2 1	---------→ 2^2
8	4 2 1	---------→ 2^3
16	8 4 2 1	---------→ 2^4
32	16 8 4 2 1	---------→ 2^5
64	32 16 8 4 2 1	---------→ 2^6
128	64 32 16 8 4 2 1	---------→ 2^7
.		.
.		.
.		.
.		
.		
.		
.		
N	....	---------→ 2^k

We know that: $N = 2^K$

Apply Log_2 both side,

$\text{Log}_2 N = \text{Log}_2 2^K$

$\text{Log}_2 N = K. \text{Log}_2 2$

$K = \text{LogN}$

Animated Image Source: Pinjee.com

Time Complexity of Binary Searching Algorithm

Best Case Time Complexity of Binary Search: O(1)

Average Case Time Complexity of Binary Search: O(logN)

Worst Case Time Complexity of Binary Search: O(logN)

Space Complexity of Binary Search: O(1) (for iterative)

Auxiliary Space

Structure

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
} [one or more structure variables];
```

Title

Author

Subject

Book ID

- Structure is another **user defined data type** that allows to combine data items of different kinds.
- Structures are **used to represent a record**. It helps us to keep track of our books in a library.

Structure

Each variable in the structure is known as a **member** of the structure.

```
struct [structure tag]
{

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}book1, book2;
                or
struct Books book1; //declaration of variables
struct Books book2;
```

Title

Author

Subject

Book ID

Visualization of Structure Variables

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
```

```
struct Books book1; //declaration of variables
```

book1

char title[50];	**| (50 bytes)**
char author[50];	| (50 bytes)
char subject[100];	| (100 bytes)
int book_id;	| (4 bytes)

2000

Total Space: 204 bytes

Accessing Structure Variables

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}book1, book2;
```

```
struct Books book1, book2; //another way

Book1.title= “Data Structure";

 Book1.author=“Yashwant Kanetkar";

 Book1.subject= “DS Tutorial";

 Book1.book_id = 6495407;
```

```
cout<< "Book 1 title : %s\n", Book1.title”;
cout<< “Book 1 author : %s\n", Book1.author”;
```

Structure as a Function Argument

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
```

```
/* function declaration */
void printBook( struct Books book );
```

```
/* function definition */
void printBook( struct Books book)
{
cout<<"Book 1 title : %s\n", book.title;
cout<<"Book 1 author : %s\n", book.author;
}
```

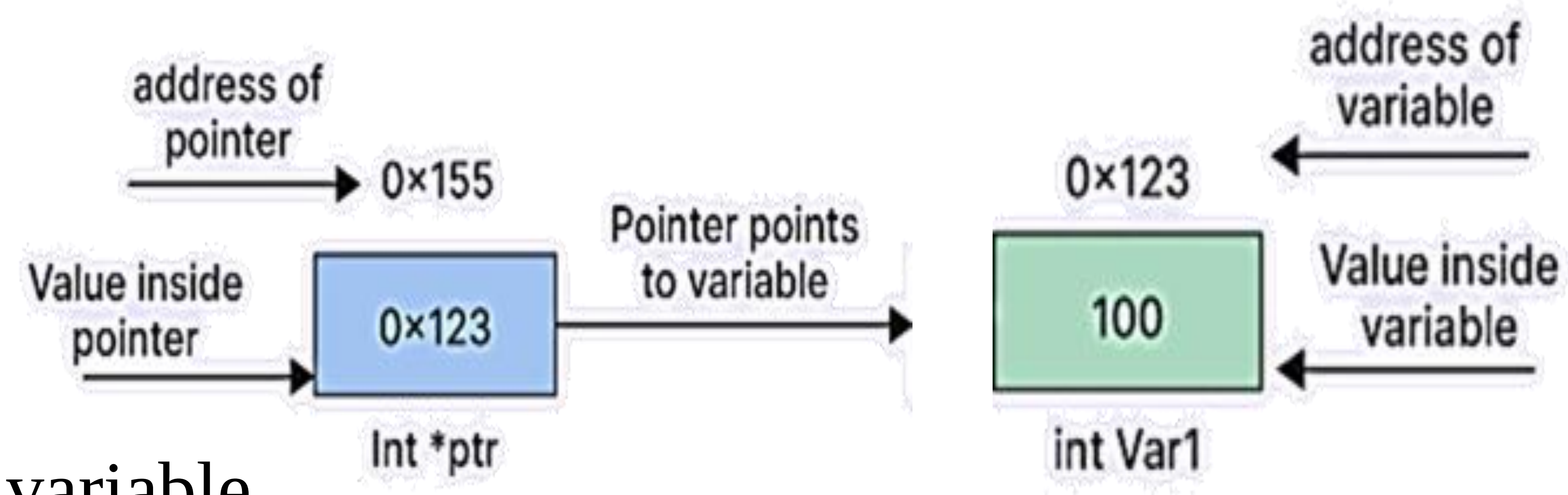
Pointer

A pointer is a variable whose **value is the address of another variable**, i.e., it stores the direct address of the memory location.

```
type *var-name; //syntax of pointer variable declaration

int    *ip;

double *dp;

float  *fp;

char   *ch ;
```

Pointer

A pointer stores the direct address of the variable, or it points to the memory location of variable.



int var1 = 100;

int *ptr ; //pointer to an integer

ptr= &var1; //store the address of int variable

cout<<ptr; //print the address of var1

cout << *ptr; // Output: 10 (value stored at the address in ptr)

* is also used as dereferencing.

Applications of Pointer

- Accessing memory directly.
- Pointers are crucial for managing memory dynamically, especially in situations where the size of data is not known in advance.
- Efficient array and structure handling.

Pointer to an Array

Array and Pointer Relationship:

- Arrays and pointers are closely related.

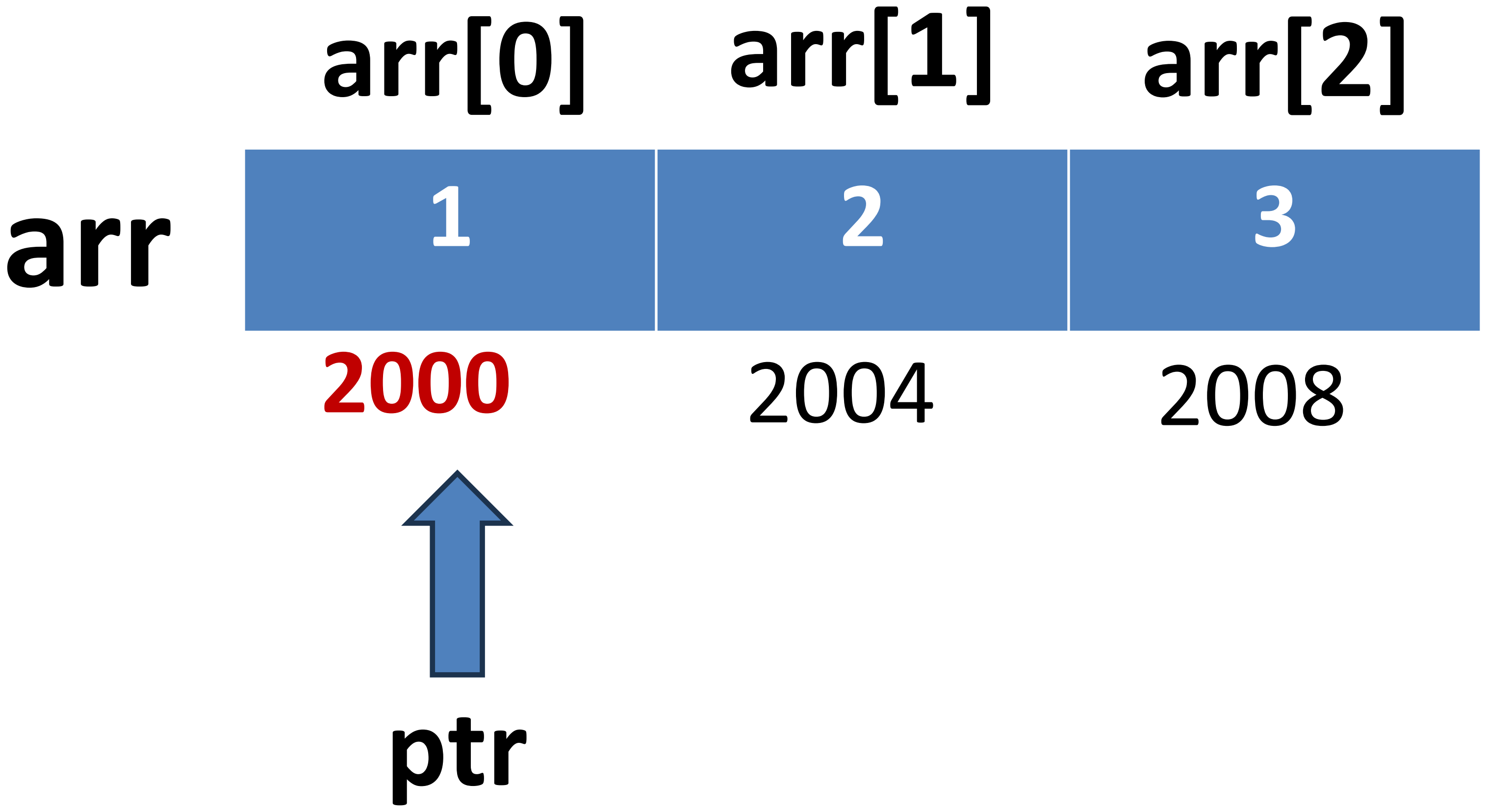
```
int arr[3] = {1, 2, 3};
```

```
cout<<arr;
```

(**arr** indicated the base address of array)

```
int* ptr = arr;  // ptr points to the first element of arr
```

(the array name acts as a pointer to the first element)



	arr[0]	arr[1]	arr[2]
arr	1	2	3
	2000	2004	2008

ptr

Pointer to an Array

Array and Pointer Relationship:

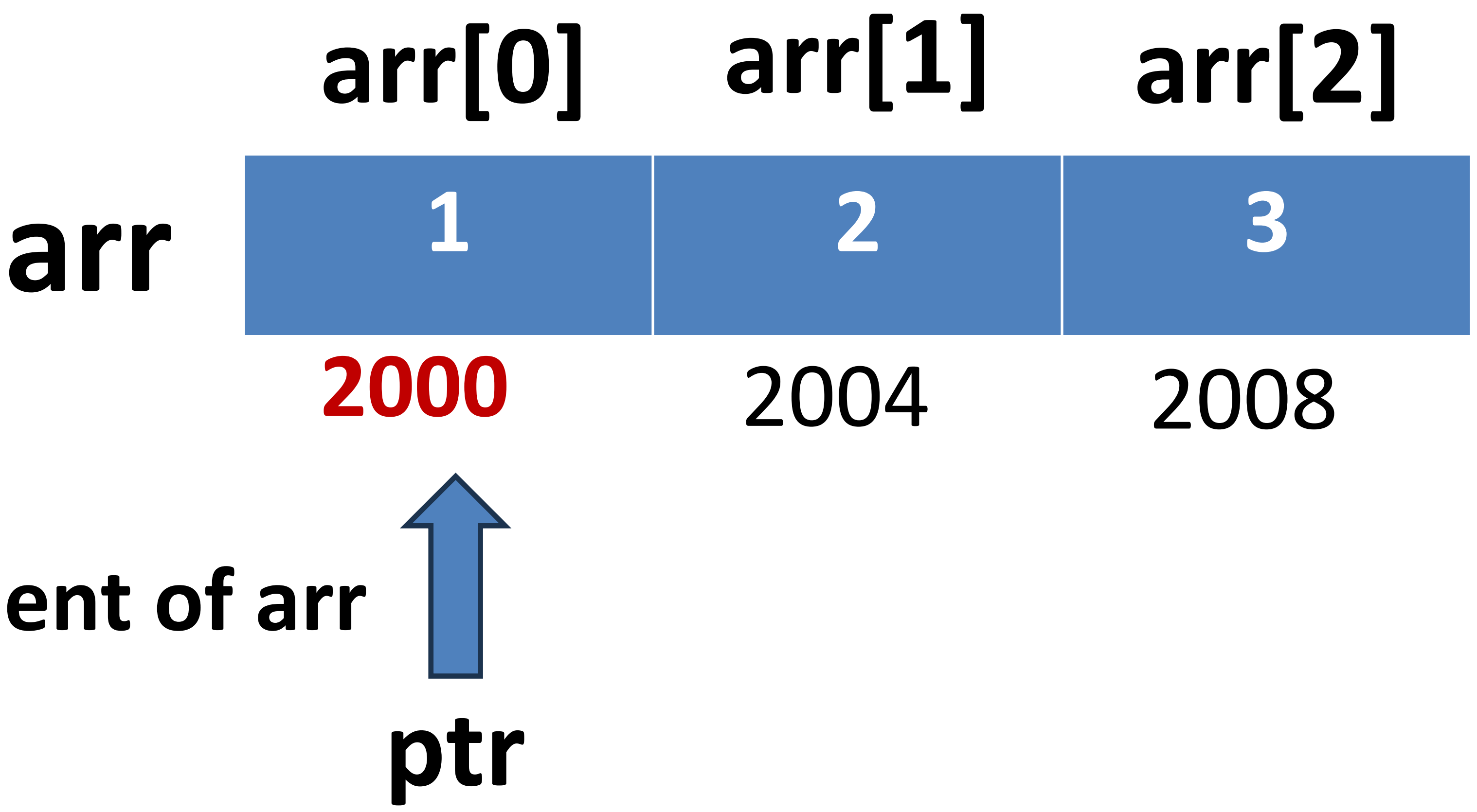
- Arrays and pointers are closely related.

```
int arr[3] = {1, 2, 3};

int* ptr = arr;  // ptr points to the first element of arr
```

Pointer Arithmetic

```
cout << *(ptr + 1);
```

// **Output: 2** (Accessing the second element)



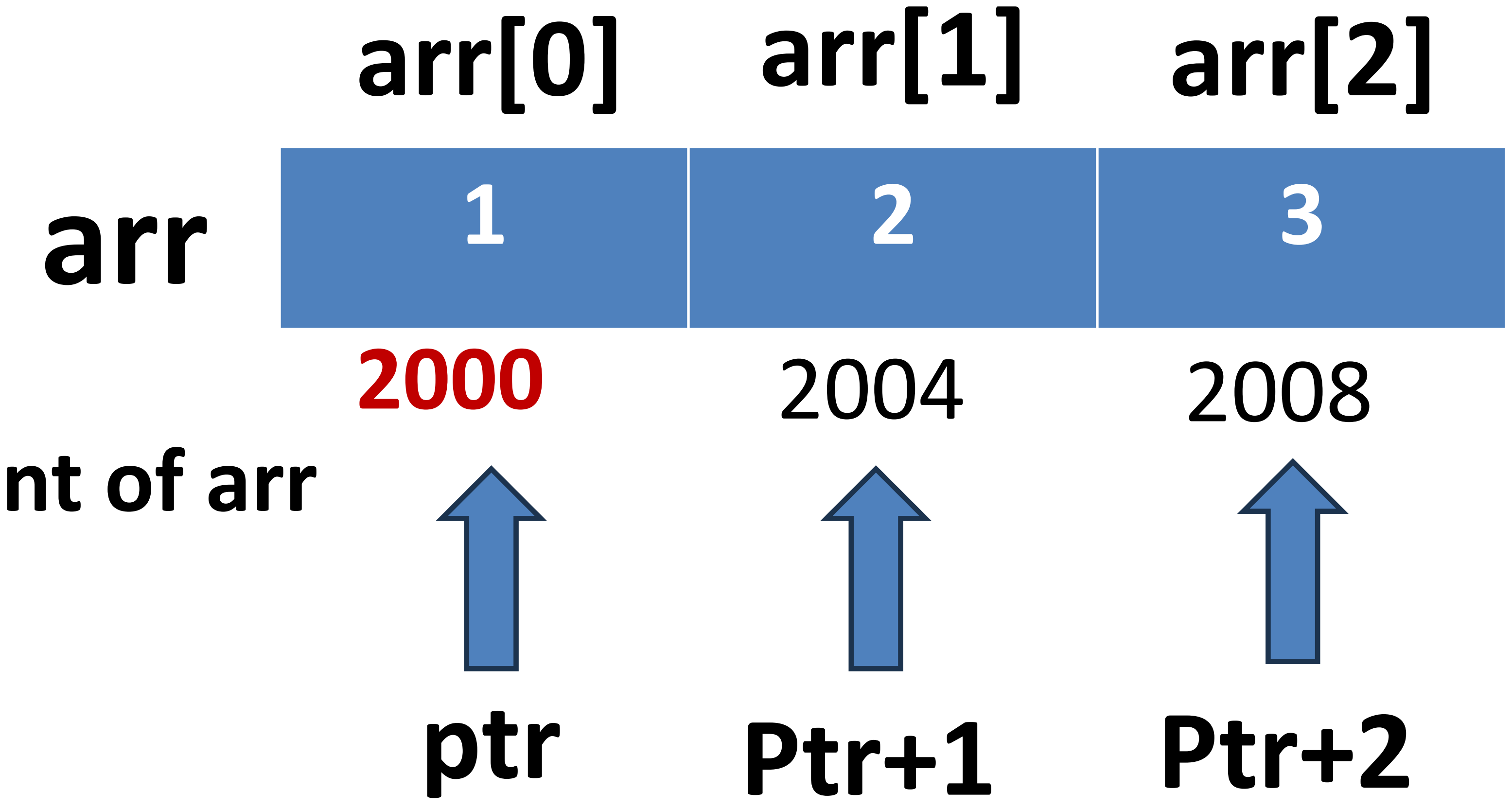
	arr[0]	arr[1]	arr[2]
arr	1	2	3
	2000	2004	2008

ptr

Pointer to an Array

Array and Pointer Relationship:

```
int arr[3] = {1, 2, 3};

int* ptr = arr;  // ptr points to the first element of arr
```



arr[0] we can write as *ptr

arr[1] we can write as *(ptr+1)

arr[2] we can write as *(ptr+2)

Applications of Pointer

- Pointers allow functions to modify variables outside their scope.

Example: Swapping of Two Numbers (Normal Approach)

```
int main()

{

int a = 5, b = 10, temp;

cout << "Before swapping." << endl; cout << "a = " << a << ", b = " << b << endl;

temp = a;

a = b;

b = temp;

cout << "\nAfter swapping." << endl; cout << "a = " << a << ", b = " << b << endl;

return 0; }
```

Applications of Pointer

- Pointers allow functions to modify variables outside their scope.

Example: Swapping of Two Numbers (using Function)

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

a	b
10	5

```
int main()
{
    int x = 5, y = 10;
    swap(x, y);
    cout << "x = " << x << ", y = " << y;
}
```

x	y
5	10

Output?

x	y
?	?

Applications of Pointer

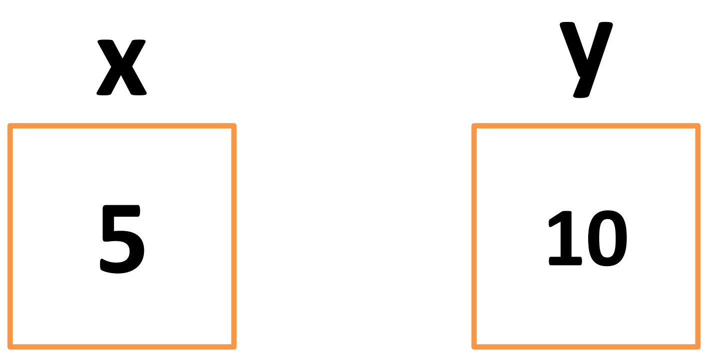
- Pointers allow functions to modify variables outside their scope.

Example: Swapping of Two Numbers (using Function)

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main()
{
    int x = 5, y = 10;
    swap(x, y);
    cout << "x = " << x << ", y = " << y;
}
```

Output:

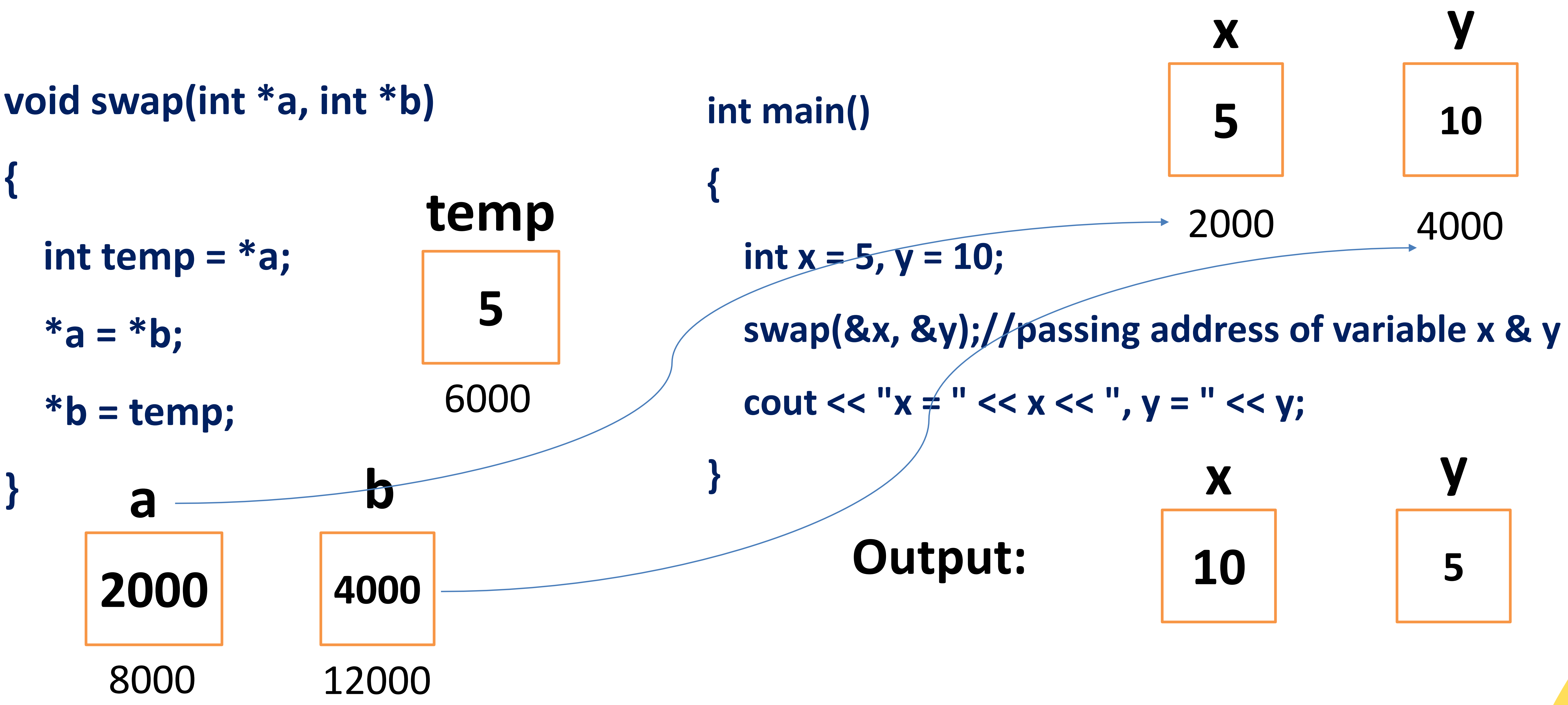


Note: When Swap function is called, it creates a local copy of variables, and scope of the variable is till function alive. the actual value of variables in the main function will remain unchanged.

Applications of Pointer

- Pointers allow functions to modify variables outside their scope.

Example: Swapping of Two Numbers (using Pointer Approach)



```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main()
{
    int x = 5, y = 10;
    swap(&x, &y);//passing address of variable x & y
    cout << "x = " << x << ", y = " << y;
}
```

Output:

Pointers to Structures

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}book1, book2;
```

```
struct Books *struct_pointer;

struct_pointer = &book1;
```

To access the members of a structure using a pointer to that structure, you must use **the → operator** as follows

struct_pointer->title;

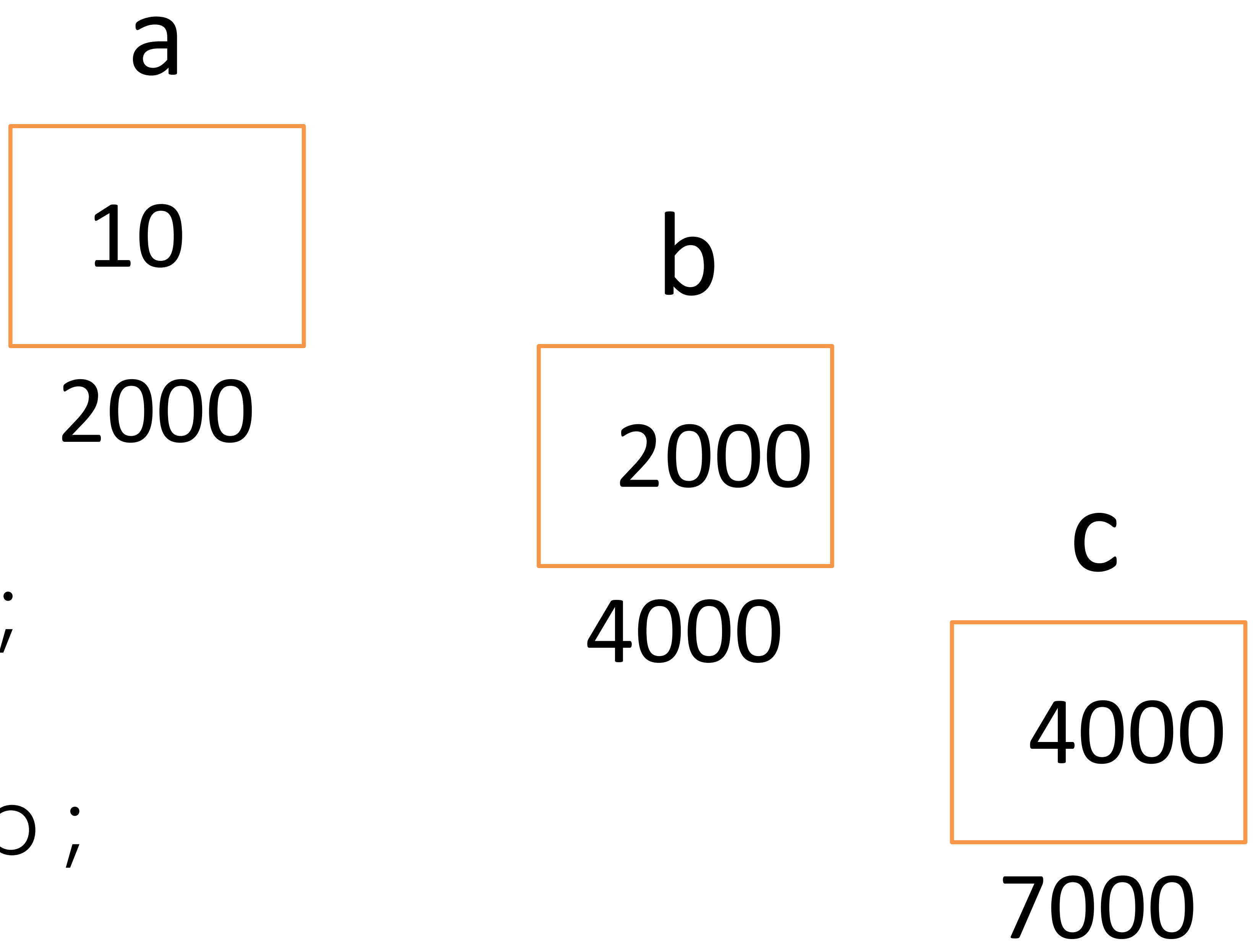
cout<< "Book title :“ << **struct_pointer**->title;

In Normal Structure

book.title

Extended Pointer (Pointer to Pointer or Double Pointers)

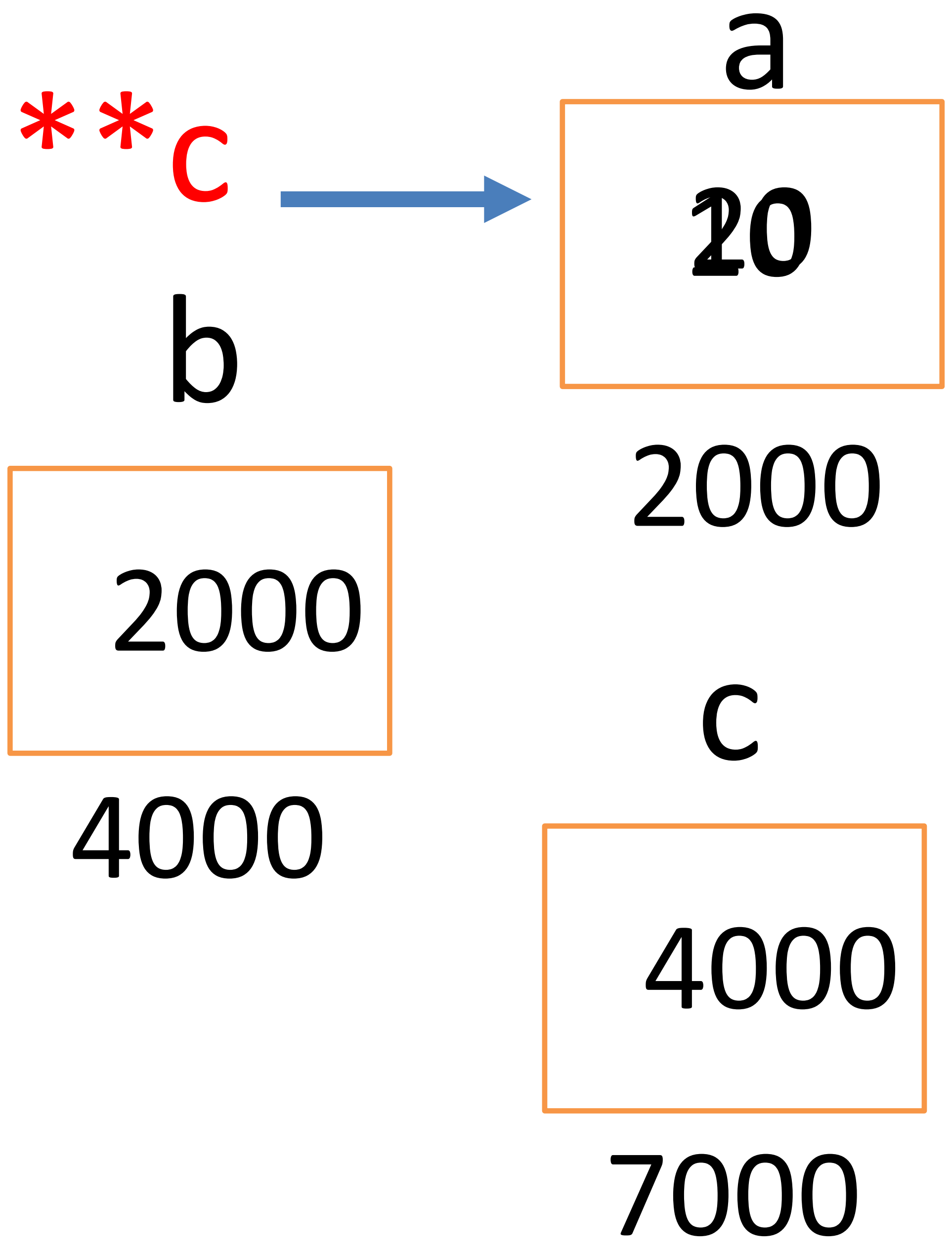
```
int  a, *b, **c;
a=10;
b=&a;
c=&b;

cout<<"The value of a is : “<< a;

cout<<"The value of b is : “<< *b ;

cout<<"The value of c is : “<< **c;
```



Output: 10, 10, 10

Assessment Time

```
int  a, *b, **c;
a=10;
b=&a;
c=&b;
**c=20;
cout<<"The value of a is : "<< a;
```



Output:

(A) 10

(B) 20

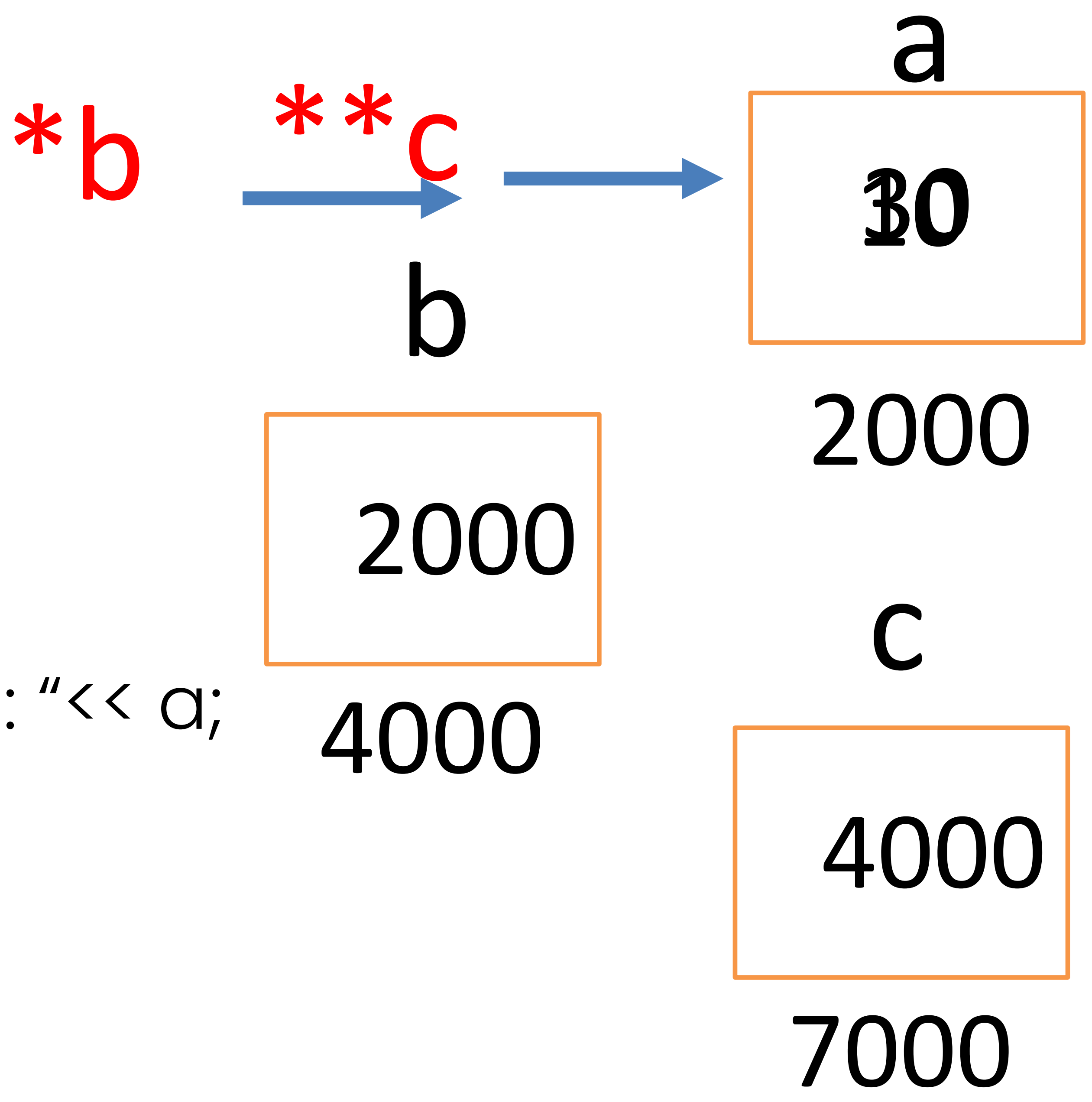
(C) 30

(D) Error

Correct Answer is **B**

Assessment Time

```
int  a, *b, **c;
a=10;
b=&a;
c=&b;
**c=a + 2*(*b);
cout<<"The value of a is : “<< a;
```



a

10

30

2000

*b

**c

b

2000

4000

c

4000

7000

Output:

(A) 10

(B) 20

(C) 30

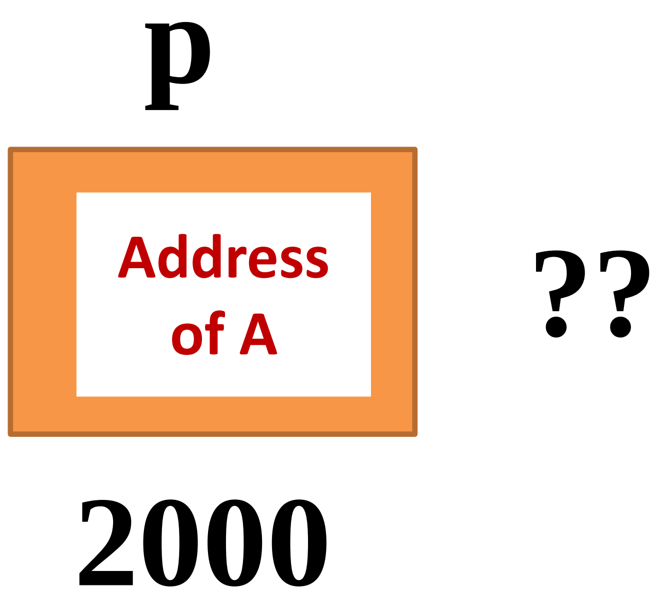
(D) Error

Correct Answer is **C**

Returning a Pointer (Assessment Time)

```
int* fun()
{
    int A = 10;
    return (&A);
}
```

```
main()
{
  int* p;
  p = fun();
}
```



p

Address of A

??

2000

Output ??

Segmentation Error

Note: Scope/Life of a Variable is limited to the Local Function only

Returning a Pointer

```
int* fun()
{
 static  int A = 10;
    return (&A);
}
```

```
main()
{
  int* p;
  p = fun();
}
```

Pointer - Homework

Explore the different operations on Pointer

Dynamic Memory Allocation (DMA)

- **Memory Allocation:** Refers to the process of reserving memory space during program execution.

Types

- **Static Memory Allocation:** Memory size is determined at compile time.
- **Dynamic Memory Allocation:** Memory is allocated during runtime.

Why DMA ?

- Flexibility in memory usage.
- Efficient use of memory.
- Allows the creation of data structures like linked lists, trees, etc., which require dynamic resizing.

How we create DMA ?

- Using **new** and **delete** Operators

  - **new** **Operator:** Allocates memory on the heap and returns a pointer to it.

  - **delete** **Operator:** Deallocates memory previously allocated with **new**.

How we create DMA ?

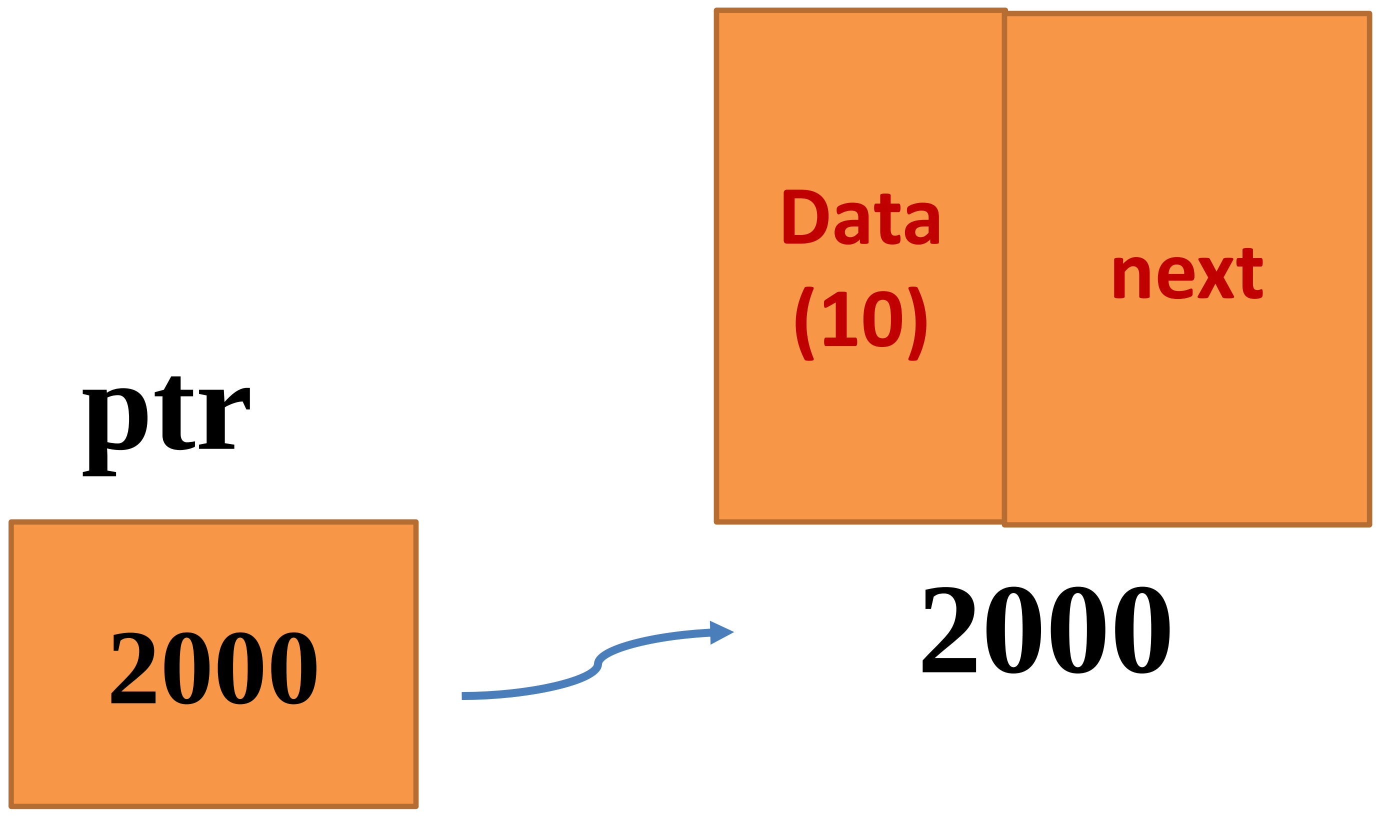
- Allocating Arrays Dynamically

```
int* arr = new int (5);  // Allocating array of 5 integers

for(int i = 0; i < 5; i++)

{

    arr[i] = i + 1;

}

delete[] arr;          // Deallocating memory
```

How we create DMA (Pointer & Structure)?

- Allocating Structure Variables Dynamically

```
struct Node {
    int data;
    Node* next;
};
```



```
Node* ptr = new Node;

ptr->data = 10;
```

```
cout << ptr->data;

// Output: 10
```

```
delete ptr;

// delete the allocated
memory
```

Explore More about DMA

Self Study

Recursion

A function calling itself is called **recursion** & corresponding function is called as **recursive function**.

T (n) = T(n-1) + n

Recursion

```
fact(int n)
{
   if (n < = 1)   // base case or stopping condition
   return 1;
   else
   return n*fact(n-1);
}
```

factorial(n) calls factorial(n-1) until n reaches 1.

Visualization of Recursion

Example - Sum of first n natural numbers

Approach(1) - Simply adding one by one

```
Algorithm Sum (A, n)
{
  S=0;
  for ( i=0; i<n; i++)
  {
   S= S + A[i];
  }
  return S;
}
```

Approach(2) – Recursive adding

Sum(n) = n+ sum(n-1)

Sum (5) = 5+4+3+2+1

Sum (5) = 5 + Sum(4)

Sum (4) = 4 + Sum(3)

Sum (3) = 3 + Sum(2)

Sum (2) = 2 + Sum(1)

Sum (1) = 1

→

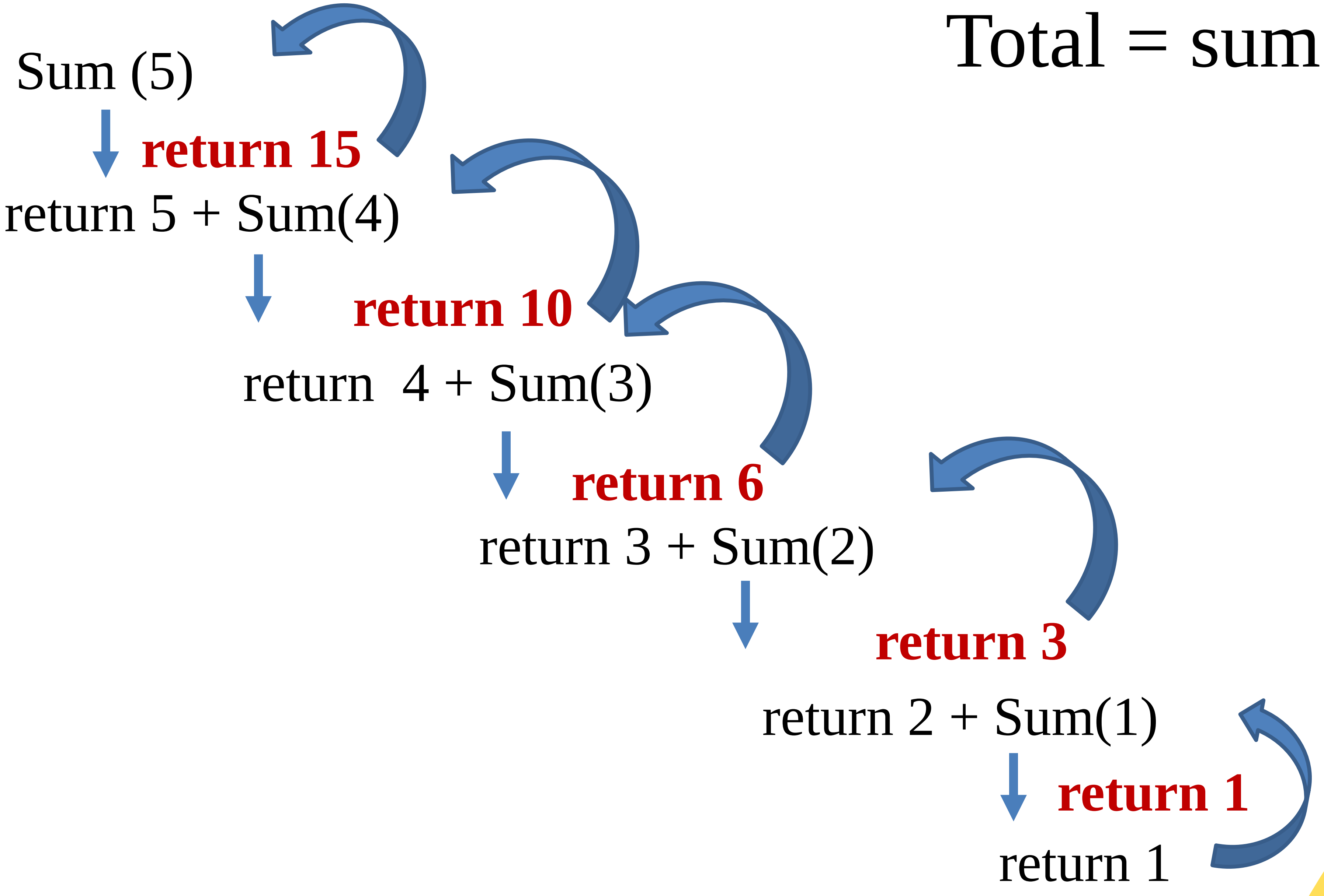
```
Algorithm Sum (n)
{
  if(n==1)
   return 1;
   else
   return  n + sum(n-1);
}
```

Visualization of Recursion

Example - Sum of first n natural numbers **Sum(n) = n+ sum(n-1)**

```
Algorithm Sum (n)
{
  if(n==1)
    return 1;
    else
    return  n + sum(n-1);
}
```

Total = sum(5) =15;



Explore these topic in detail

1. Recursion Tree
2. Call Stacks

Examples of Recursion

Fibonacci Series

```
int fibonacci(int n) {
    if (n == 0)  // Base case 1
        return 0;
    else if (n == 1)  // Base case 2
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
}
```

Each term is the sum of the previous two terms. Recursion continues until n is 0 or 1.

Examples of Recursion

Self Practice

Sum of Digits

- Write a recursive function to find the sum of the digits of a given positive integer n.

Power of a Number

- Write a recursive function to calculate x^n where x is a base and n is the exponent.

Reverse a String

- Write a recursive function to reverse a string.

Palindrome Check

- Write a recursive function to check if a given string is a palindrome.

Applications of Recursion

- Solving Mathematical Problems (Tower of Hanoi)
- Binary Search
- Tree Traversals
- Solving Complex Problems (Divide and Conquer, back Tracking…)

Recurrence Vs Iteration

- Both used to solve problems & achieve similar results, but they differ significantly in their implementation, usage, and performance.

- Recursive solutions are more expensive than corresponding iterative solutions

Explore more (Self Study)

When to Use Recursion

Use Recursion

- When the problem can be broken down into similar sub-problems.
- When the recursive structure naturally mirrors the problem (e.g., tree structures).
- When simplicity and readability are more important than performance.
- In problems like tree traversal, dynamic programming (with memoization), and backtracking (e.g., N-Queens).

Use Iteration

- When performance and memory efficiency are critical.
- When the problem involves simple repetitive tasks.
- When recursion depth could be too large, risking stack overflow.
- In problems like searching, sorting, and iteration over data structures like arrays and linked lists.

What does recursion relation mean

$$\text{Sum}(n) = \begin{cases} 1 & \text{if } n=1 \\ \text{Sum}(n-1) + n & \text{if } n>1 \end{cases}$$

← **Base/Stopping Condition**

- ✓ A recurrence relation is an equation or inequality that describes a **function in terms of its values on smaller inputs**.
- ✓ Used to **reduce complicated problems** to an iterative process based on simpler versions of the problem
- ✓ **T(n) term is used to define** the time complexity in recurrence relation analysis.

Writing Recurrence Relations

```
Algorithm Demo(n)        ———— T(n)
{
  If (n>0)
    {
     Print " n";         ———— 1
     Demo(n-1);          ———— T(n-1)
     }
}
———————————————
    T(n)= T(n-1) + 1
```

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + 1 & \text{if } n>0 \end{cases}$$

Writing Recurrence Relations

Example - **Sum of first n natural numbers**

```
Algorithm Sum (n)                    T(n)
{
  if(n==1)
   return 1;                         1
   else
   return  n + sum(n-1);             n + T(n-1)
}
```

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + n & \text{if } n>1 \end{cases}$$

Solving Recurrence Relations

There are four methods for solving Recurrence Problems

- ✓ Back Substitution/ Iteration Method
- ✓ Recursion Tree Method
- ✓ Master Method
- ✓ Substitution Method

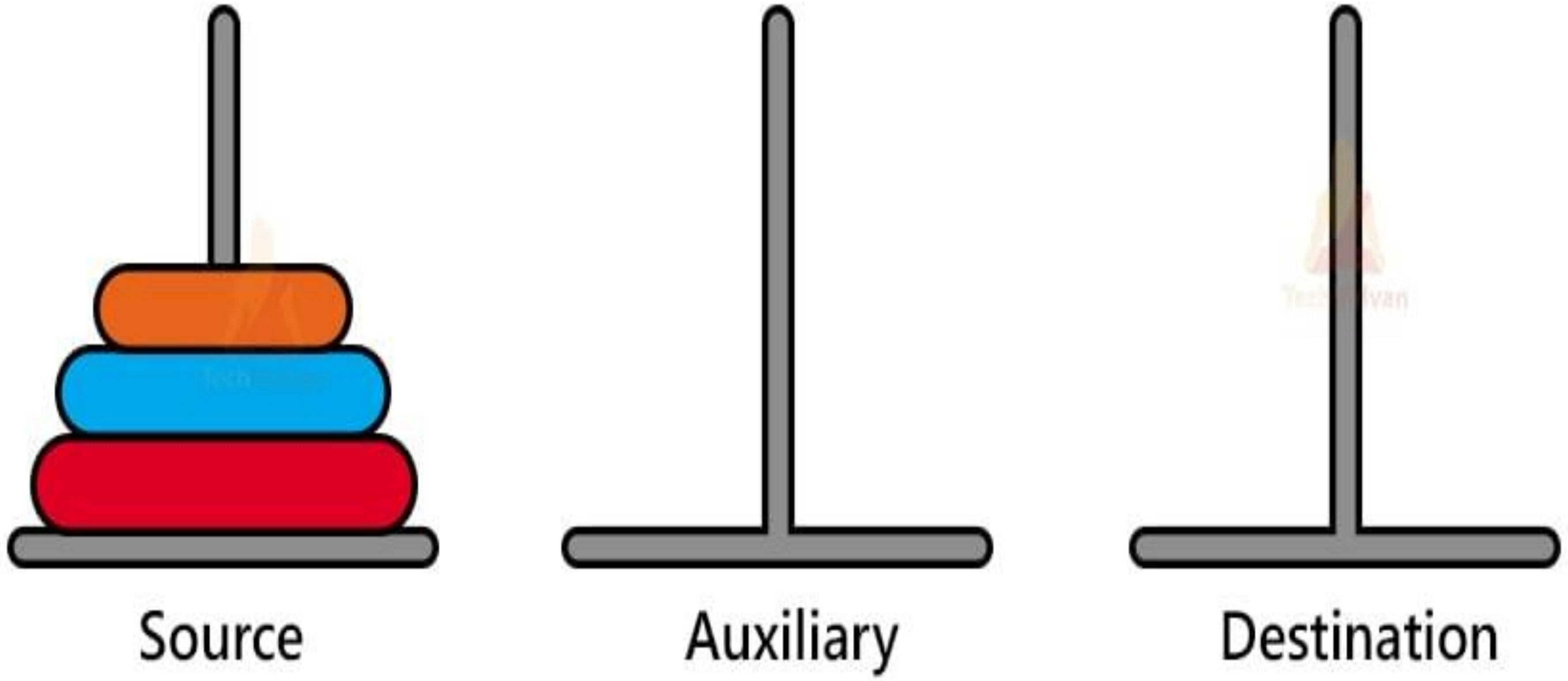
Note:

- Detail of these topic will be covered in the course Design and analysis of Algorithms (Next Semester)

Self Practice Problem using Recursion

Tower of Hanoi Problem

There are three towers, 3 disks, with decreasing sizes, placed on the first tower. You need to move all of the disks from the first tower to the last tower, A large disk can not be placed on top of a smaller disk. The remaining tower can be used to temporarily hold disks



Self Practice Problem using Recursion

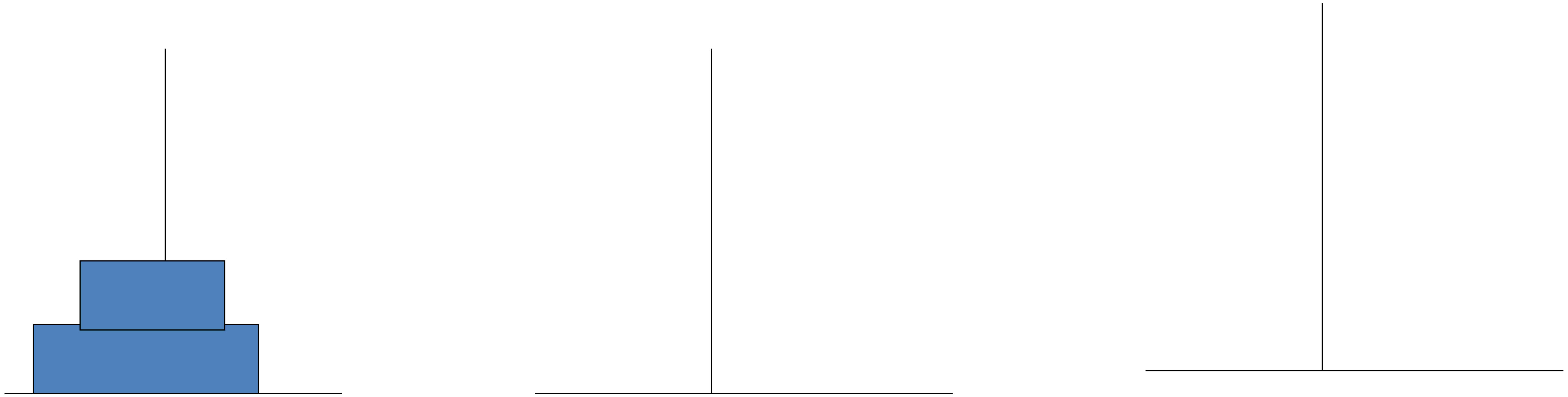
Tower of Hanoi Problem

- How many operations needed to transfer 7 disks from source tower to destination tower?

- Given that 10 disks can be moved in 1023 operations, how many operations needed for 11 disks?

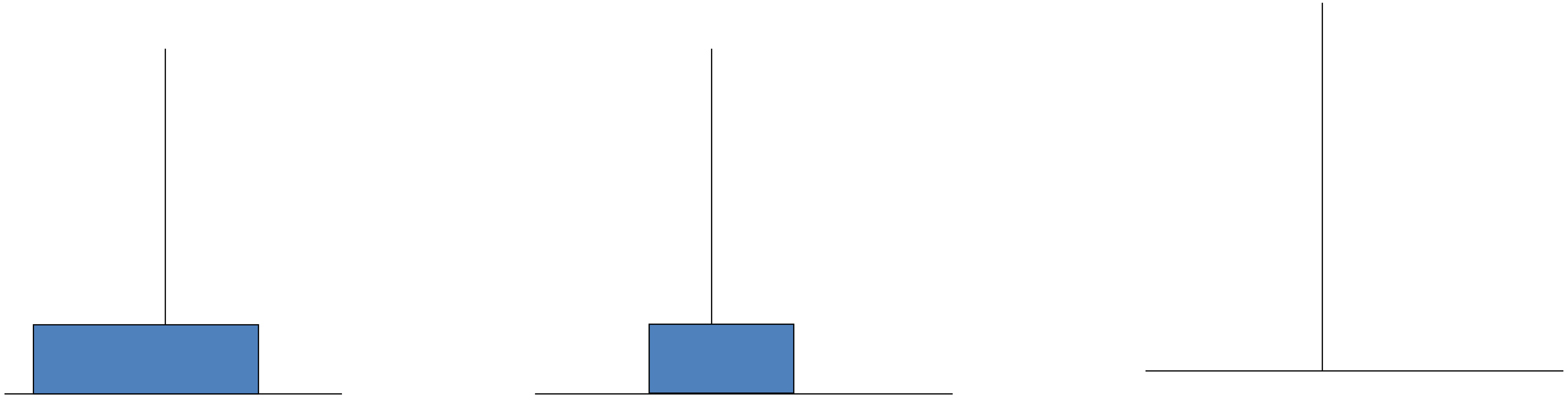
Analyzing the Tower of Hanoi Problems (2 Disks)

- The problem is to move two disks from first tower to the third tower
- The second tower is the temporary tower



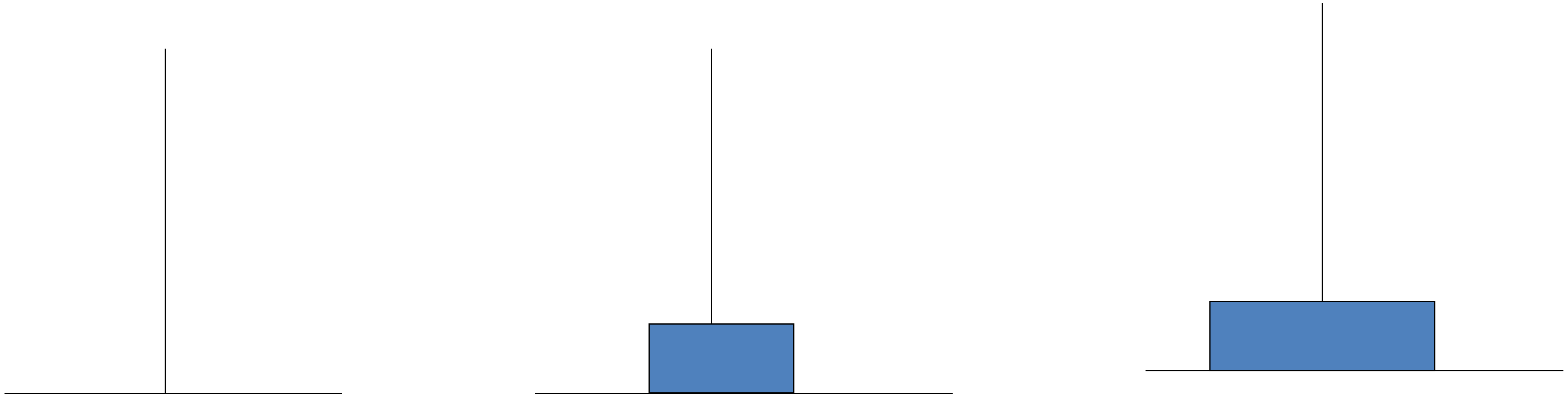
Analyzing the Tower of Hanoi Problems (2 Disks)

Step 1 Move n-1 disks source to temp



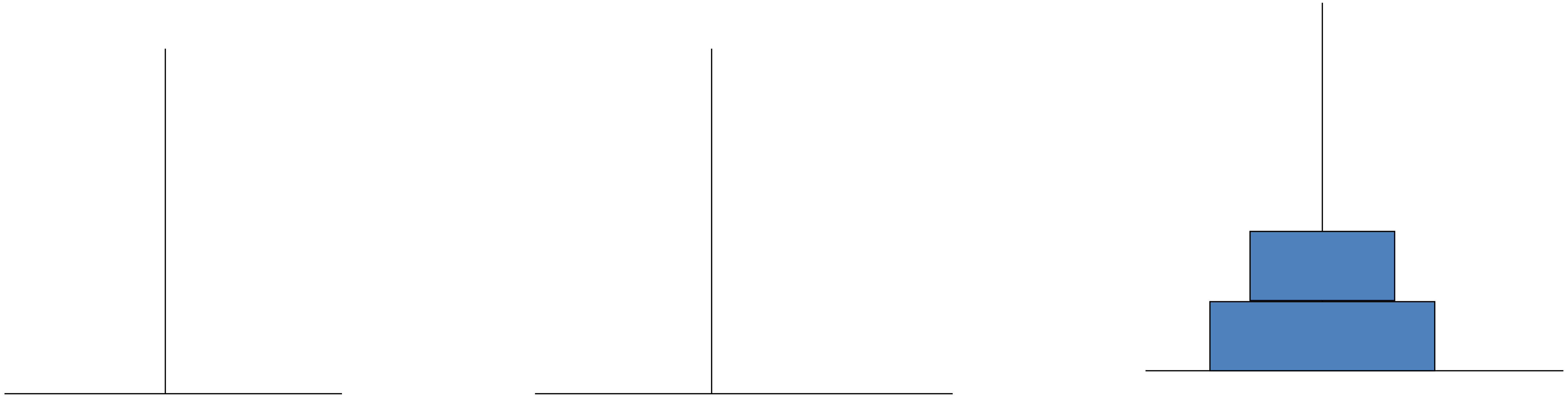
Analyzing the Tower of Hanoi Problems (2 Disks)

Step 2 Move disk 1 from source to destination



Analyzing the Tower of Hanoi Problems (2 Disks)

Step 3: Move disk from temp to destination



Analyzing the Tower of Hanoi Problems (2 Disks)

So to move 2 disks , it required 3 steps in all (Total 3 operations or movements)

(It does not matter which tower is source and which is destination.)

Now let us extend the problem to 3 disks-

We shall do it in 3 large steps

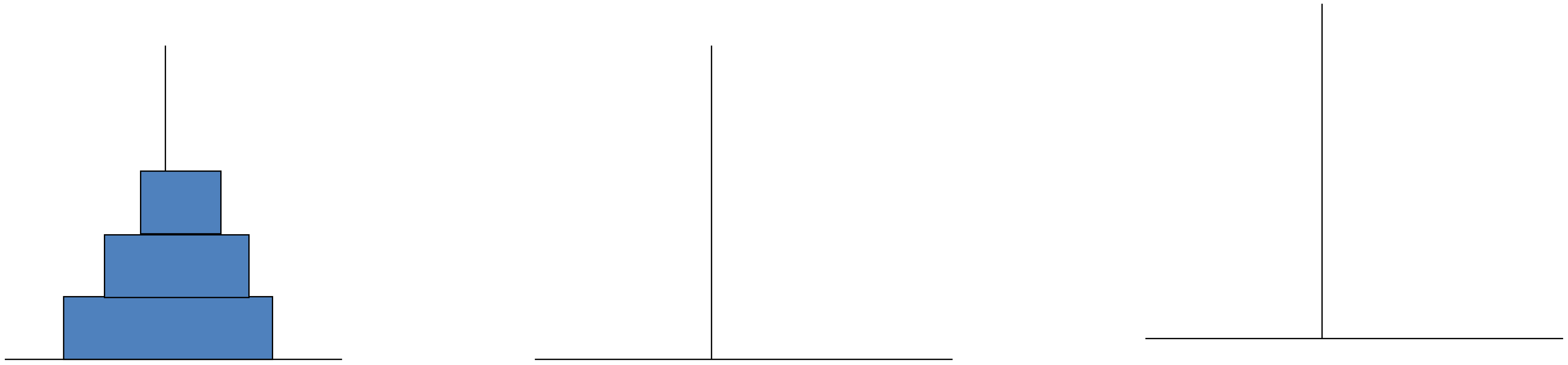
Step 1: move 2 disks from source to temp (use destination T to do this)

Step 2: Move the *last remaining disk* from source to destination

Step 3: Move 2 disks from temp to destination (using source T to do this)

Analyzing the Tower of Hanoi Problems (3 Disks)

- Let first tower be called a (source tower)
- Let third tower be called b (destination tower)
- Let second tower be called c (temporary tower)



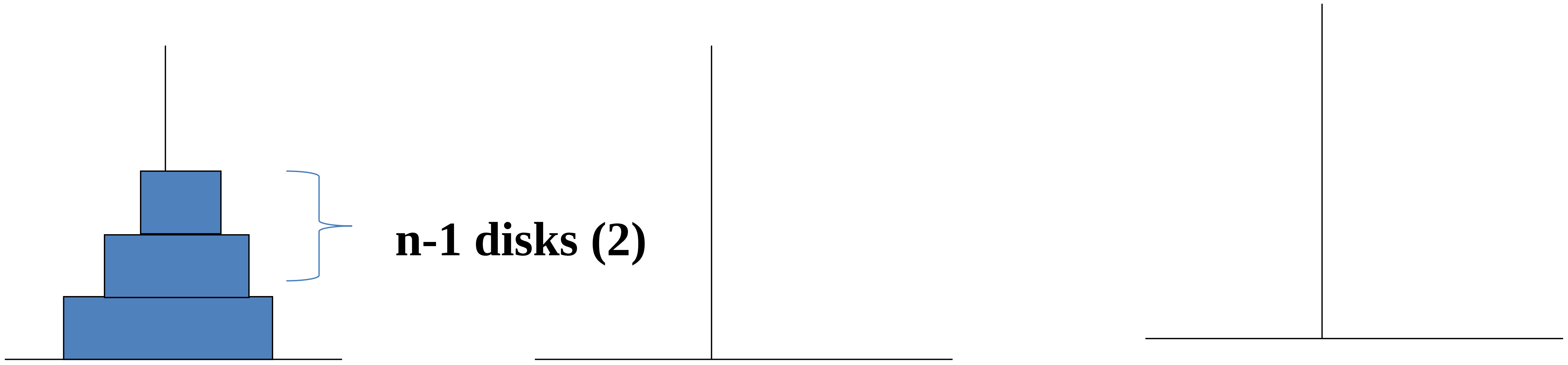
A
(source tower)

C
(temporary tower)

B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

- Remove n-1 disks to temp tower
- We already know the solution to this problem, just solved it in previous slides
- Let us simply copy the solution



n-1 disks (2)

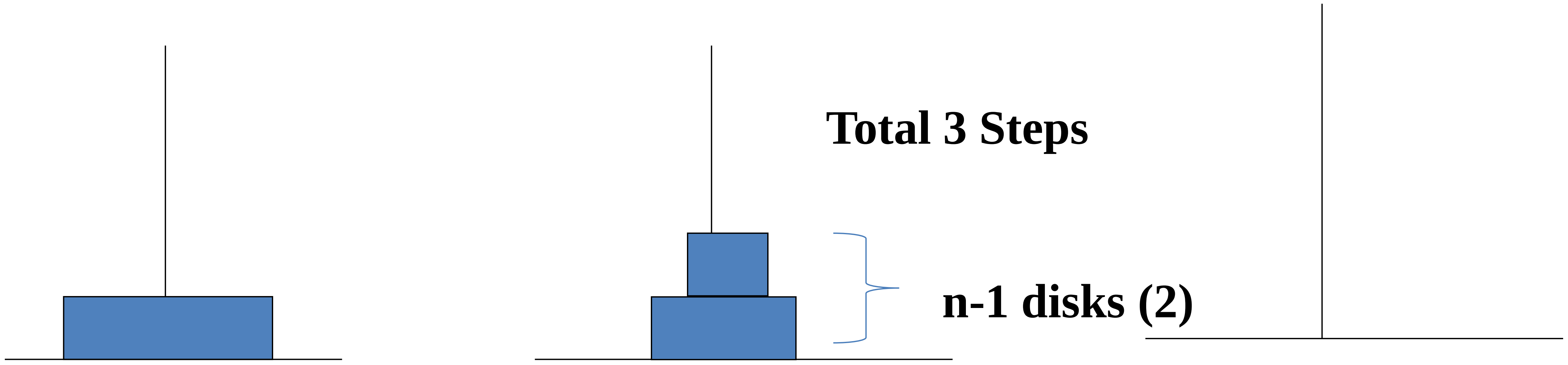
A
(source tower)

C
(temporary tower)

B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

Step 1: To move 2 disks , it required 3 steps in all (Total 3 operations or movements)



Total 3 Steps

n-1 disks (2)

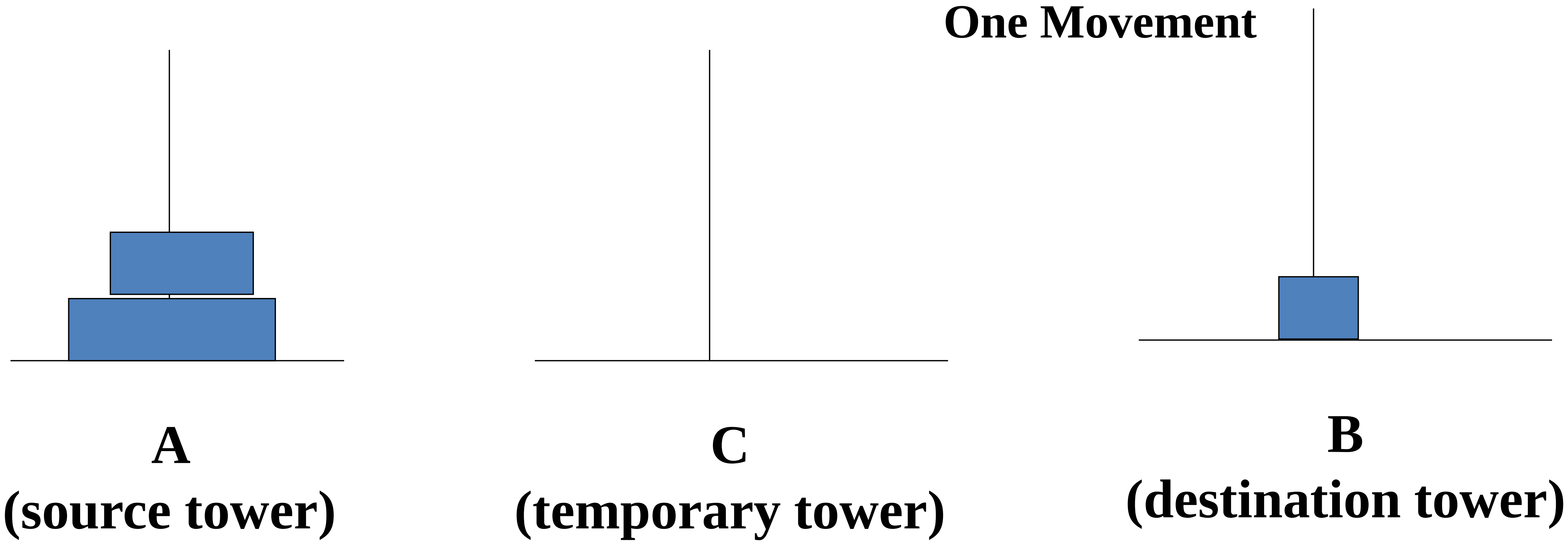
A
(source tower)

C
(temporary tower)

B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

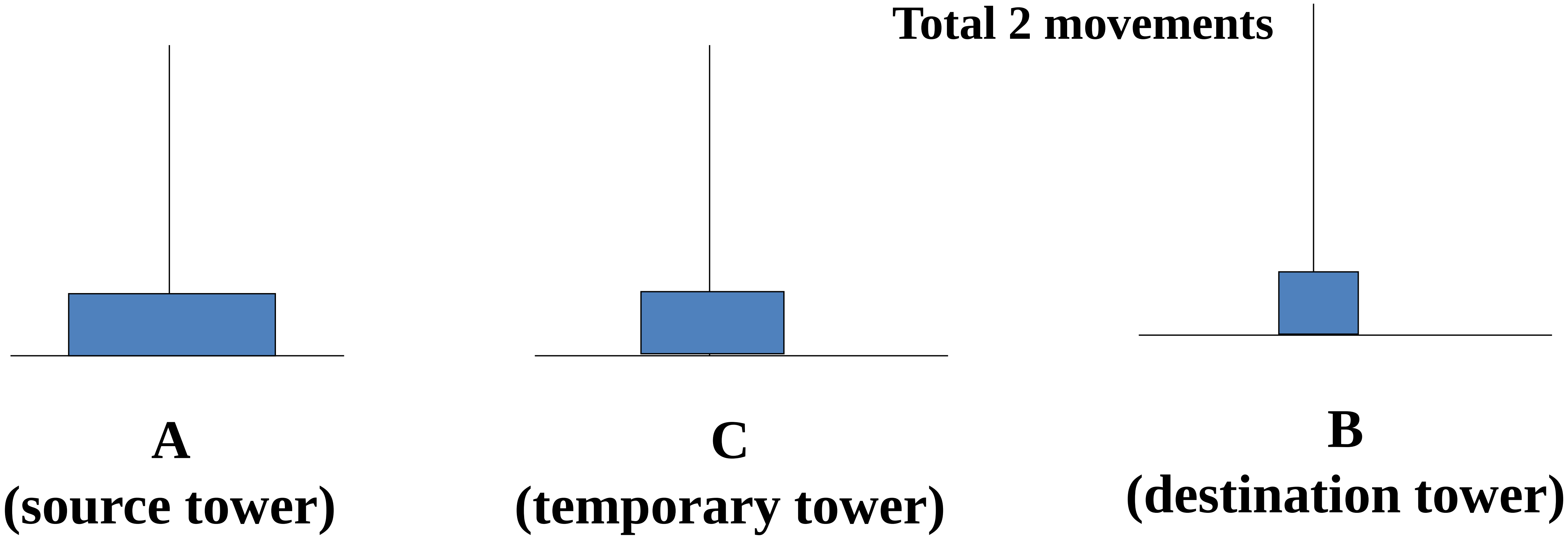
Step 1a: Start moving n-1 disks from source tower to temp tower using destination tower



Analyzing the Tower of Hanoi Problems (3 Disks)

Step 1 contd.

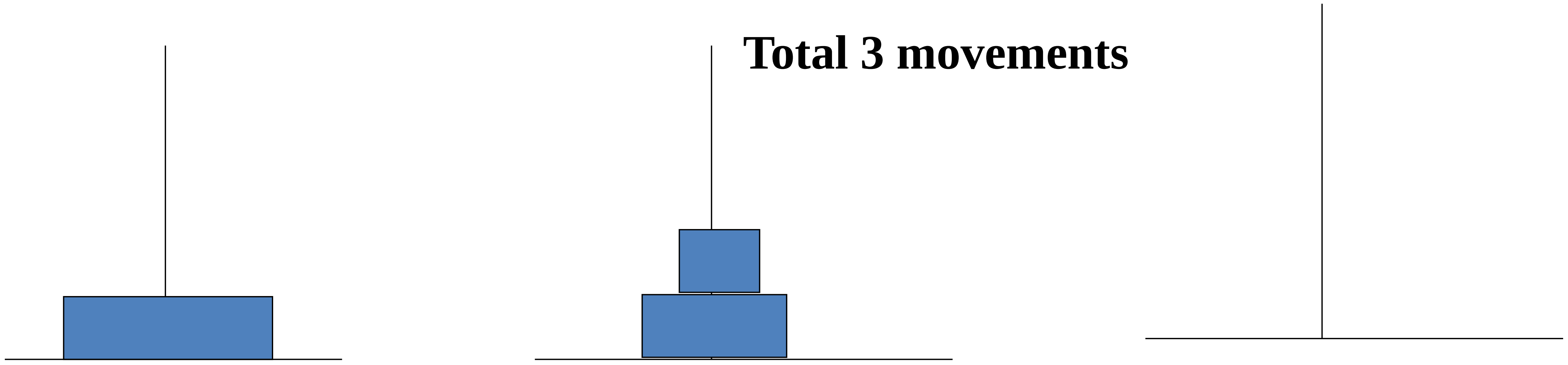
carry on process of moving n-1 disks from source tower to temp tower using destination tower



Analyzing the Tower of Hanoi Problems (3 Disks)

Step 1

We have completed moving n-1 disks from source to temp using destination **(total 3 movements),** We can go for step 2



A
(source tower)

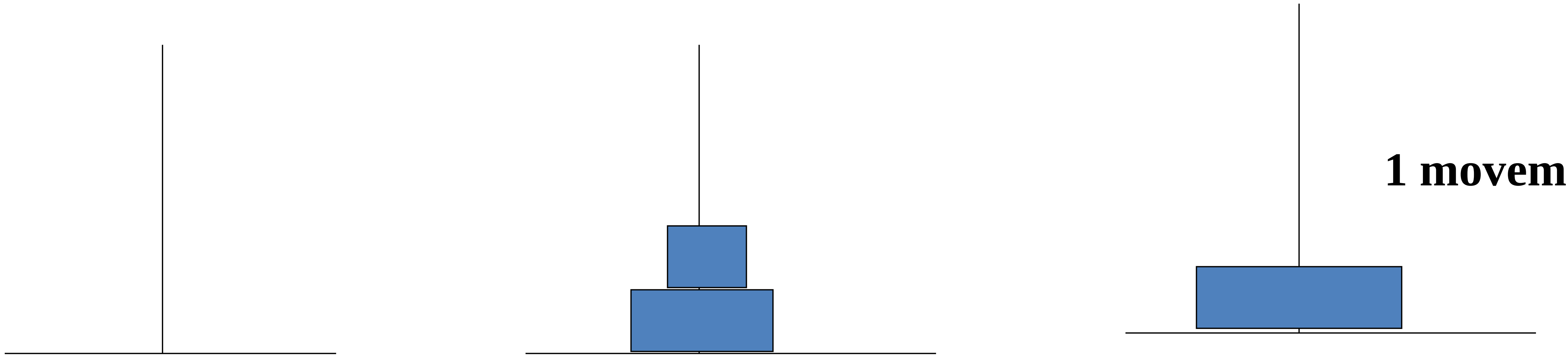
C
(temporary tower)

B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

Step 2

Move remaining disk (disk 1) from source to destination using temp (It is just a single step), Total **One Movement**



A
(source tower)

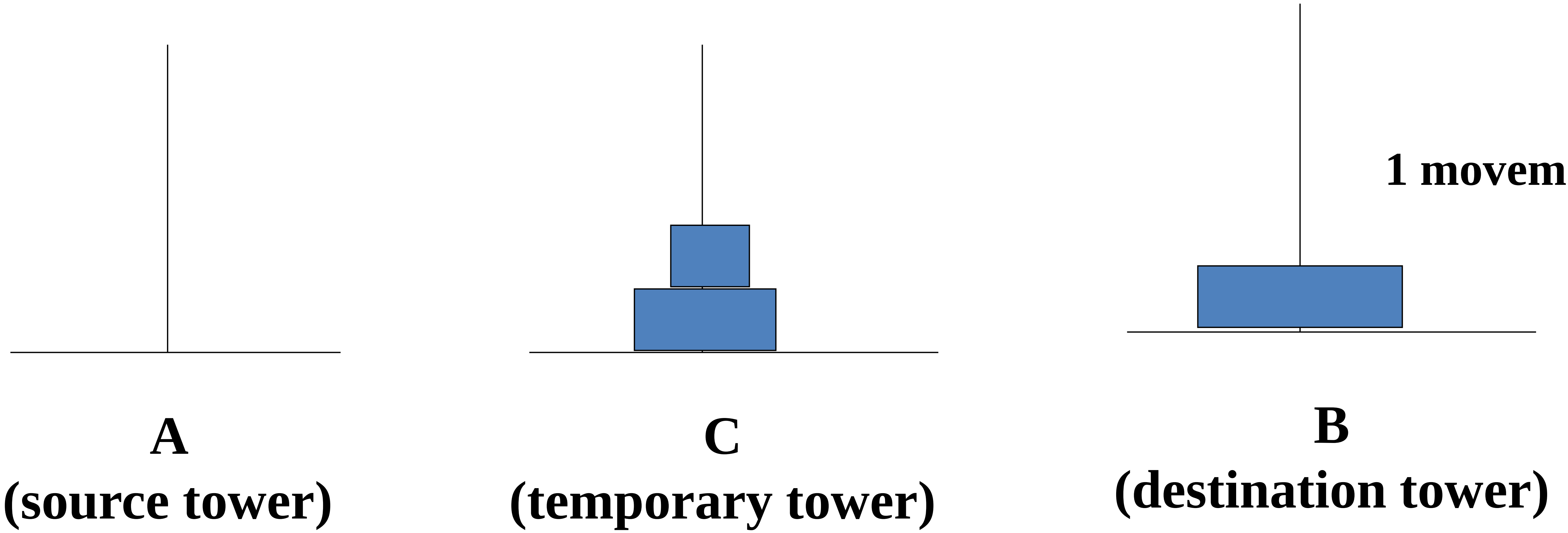
C
(temporary tower)

B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

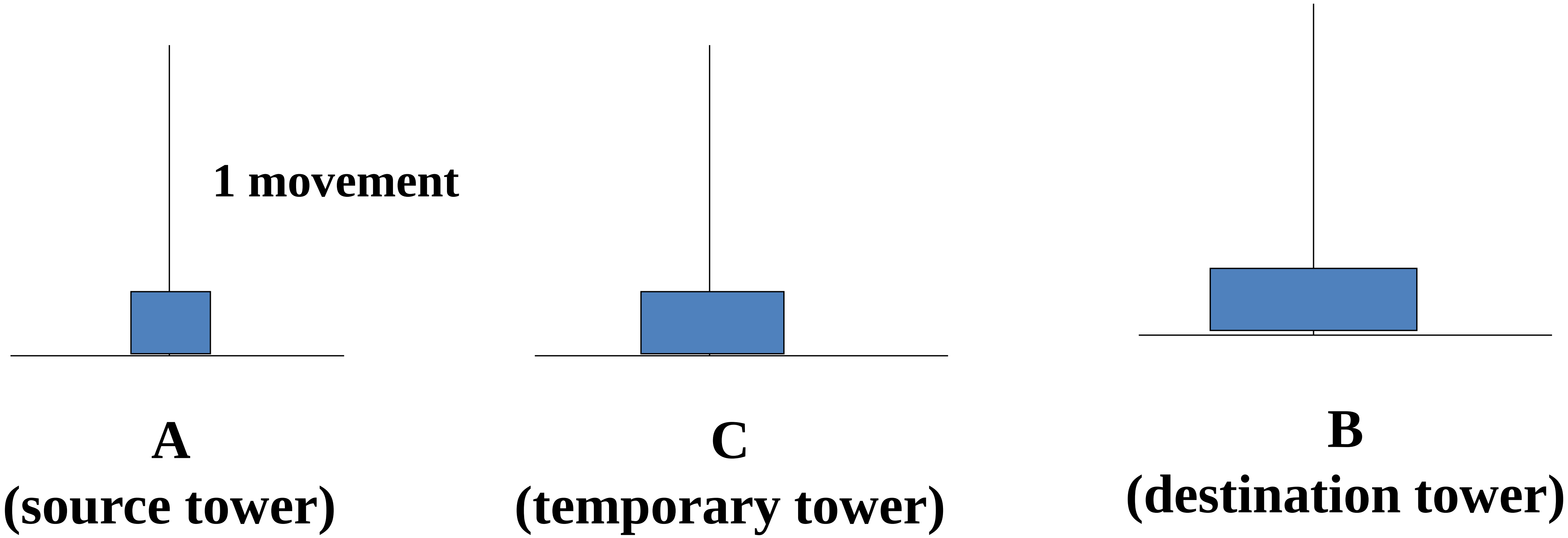
Step 3: Move n-1 disks (2 disks) from temp to destination using source tower

- We already know how to move 2 disks from one tower to any other tower



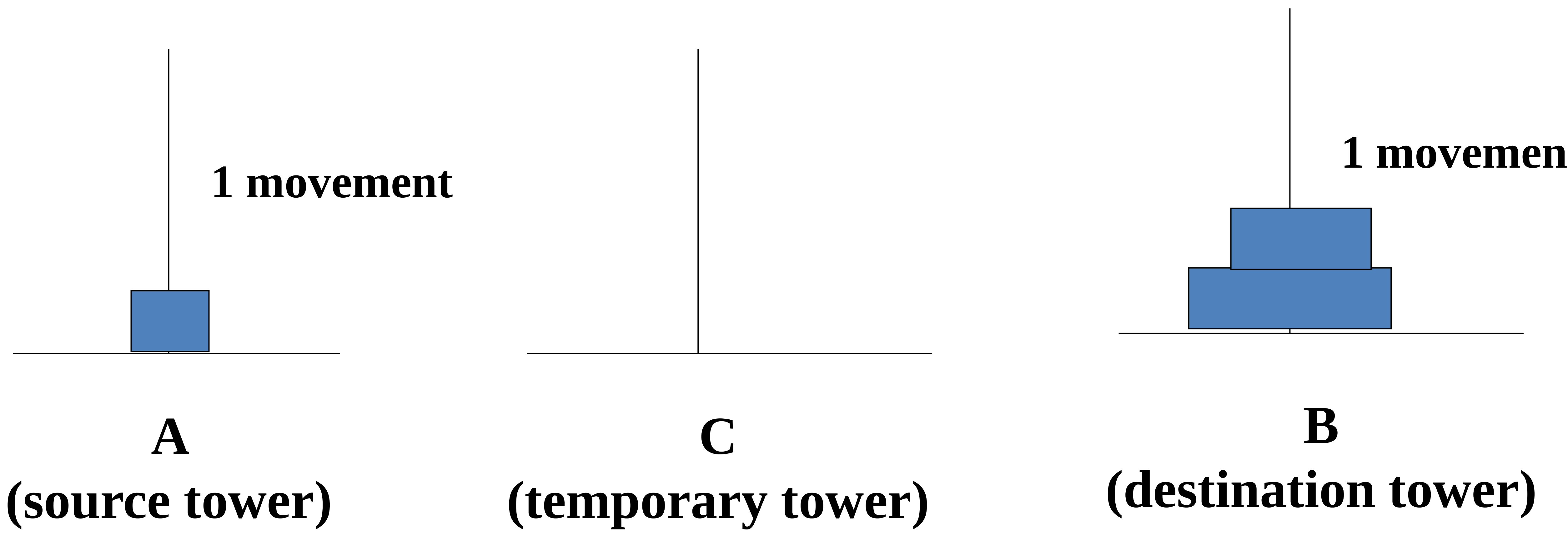
Analyzing the Tower of Hanoi Problems (3 Disks)

Step 3: Move n-1 disks (2 disks) from temp to destination using source tower



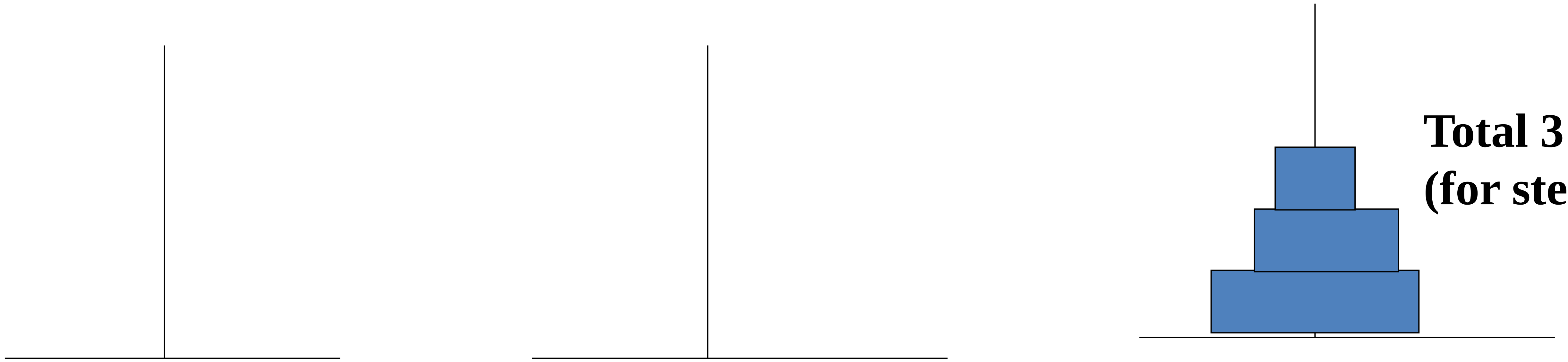
Analyzing the Tower of Hanoi Problems (3 Disks)

Step 3: Move n-1 disks (2 disks) from temp to destination using source tower



Analyzing the Tower of Hanoi Problems (3 Disks)

Step 3: Move n-1 disks (2 disks) from temp to destination using source tower



Total 3 movements (for step 3)

A
(source tower)

C
(temporary tower)

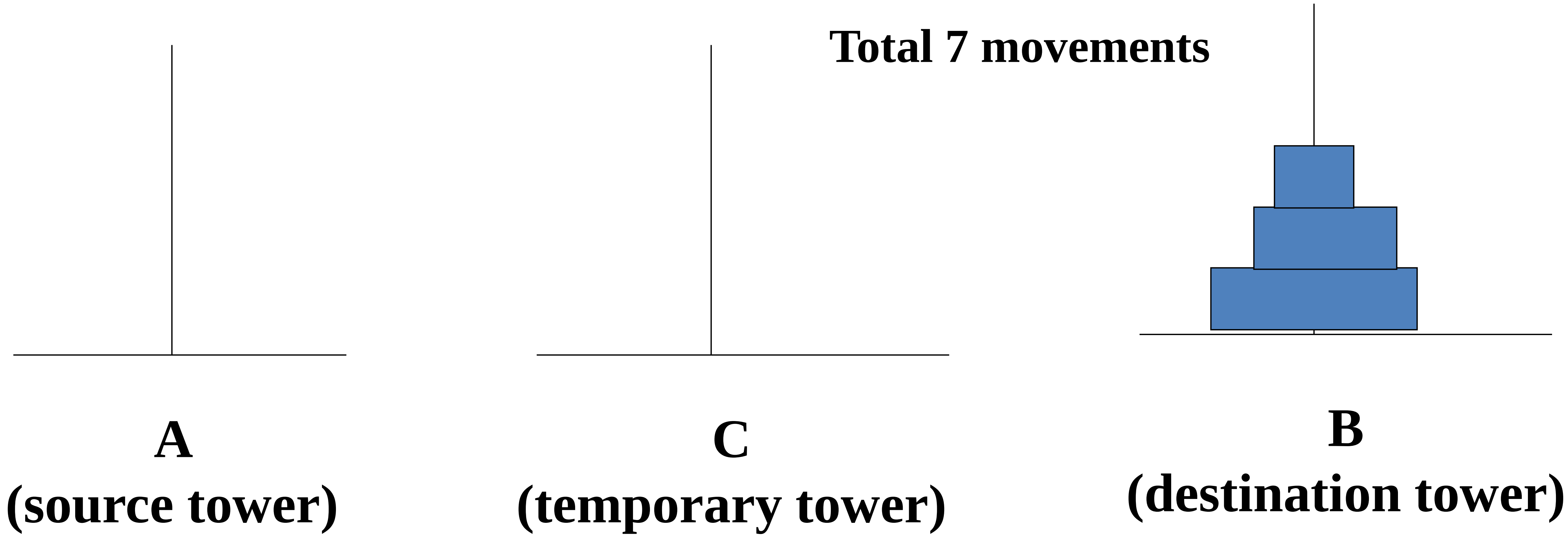
B
(destination tower)

Analyzing the Tower of Hanoi Problems (3 Disks)

We have now completed moving n-1 disks from temp to destination using source tower.

Total Movements = Step 1 + Step 2 + Step 3

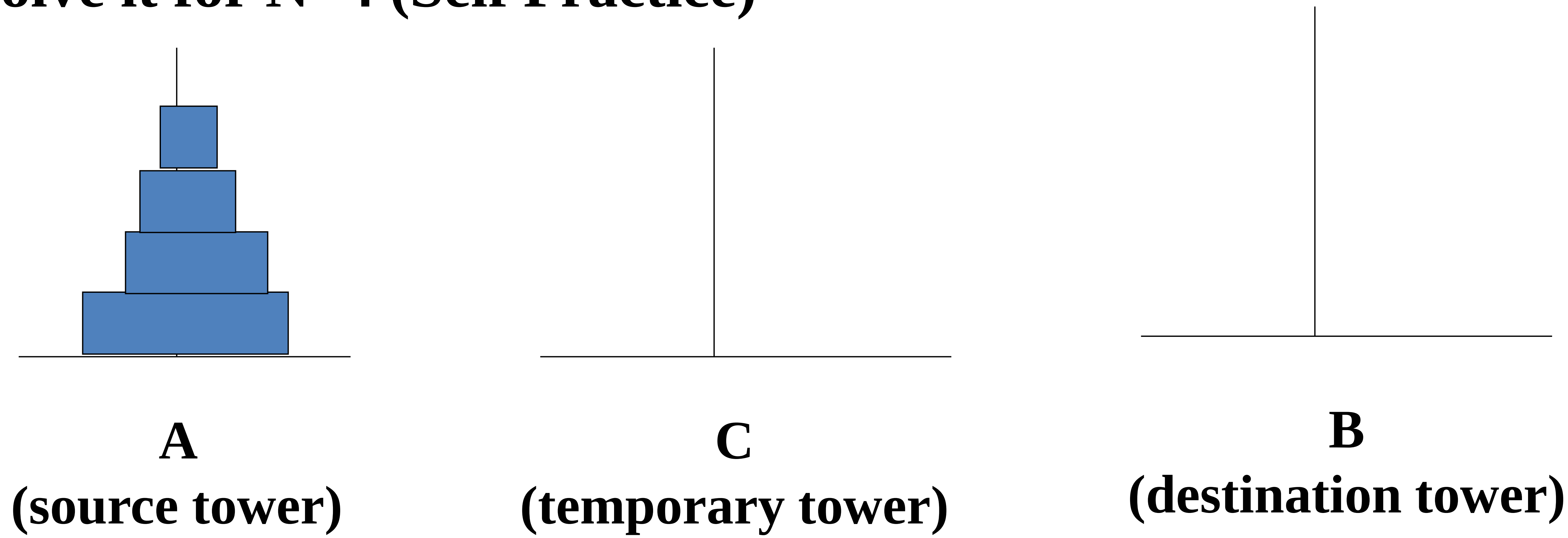
3 + 1 + 3 = 7



Analyzing the Tower of Hanoi Problems (3 Disks)

- In 3 major steps , we have solved the problem for n= 3 disks.
- We can now solve the problem for any value of n.
- The 3 steps remain the same for any size problem

Solve it for N=4 (Self Practice)



Analyzing the Tower of Hanoi Problems (n Disks)

Although there are Three major steps, the **number of actual operations (movements)** will depend on value of n.

- For n = 1, 1 step needed
- For n = 2, 1 + 1 + 1 (3 steps in 3 operations)
- For n = 3, 3 + 1 + 3 (3 steps in 7 operations)
- For n =4, 7 + 1 + 7 (15 operations)

…………..

Algorithm of Tower of Hanoi Problems (n Disks)

```
void Hanoi( n,  a,  b,  c)  /*a :source, b:destination, c:temp*/
  {
    if (n == 1)                /* base case */
      Move( a, b);
    else {                     /* recursion */
      Hanoi( n–1, a, c, b);   // a to c using b
      Move( a, b);            // last disk a to b
      Hanoi( n–1, c, b, a);   // c to b using a
         }
  }
```

Visualize it (Self Practice)

Explore Sorting Algorithms

Explore the Sorting Algorithms

- Stable and In-Place Sorting Algorithms
- Bubble
- Selection
- Insertion Sort
- Merge Sort

Any Queries?

Office MCub311
Discussion Time: 3-5 PM
Mob: +91-8586968801