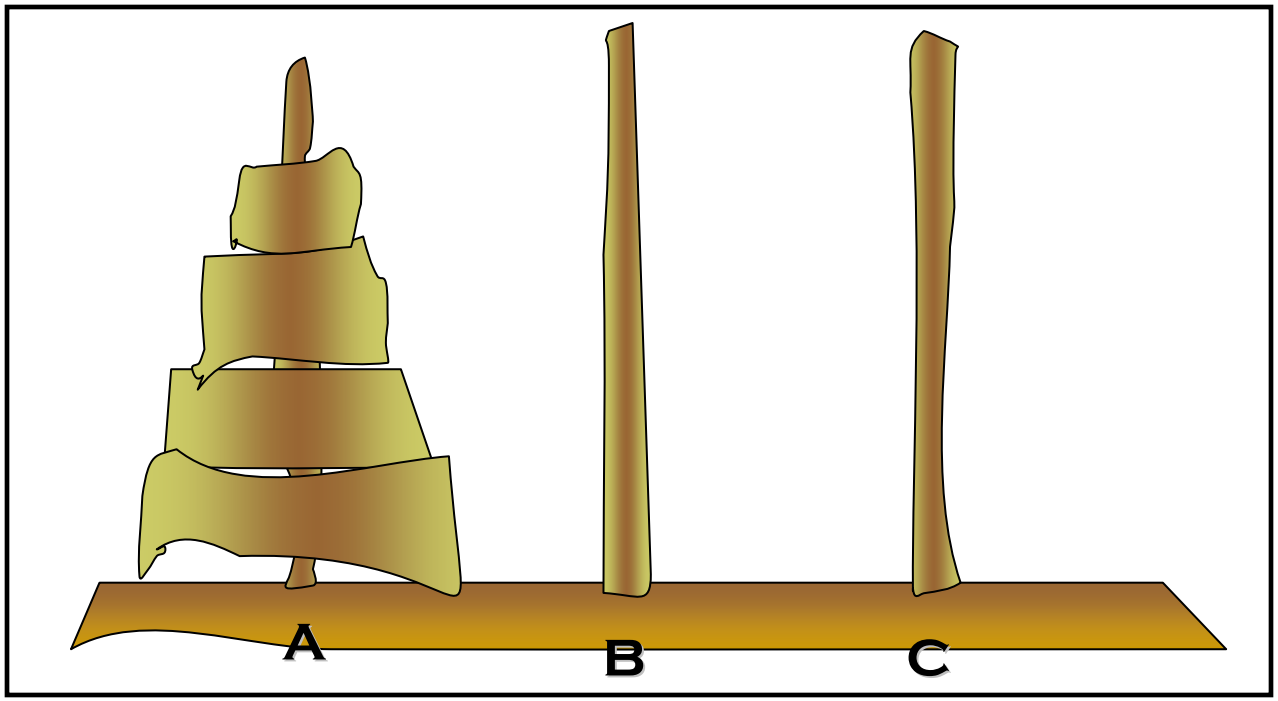
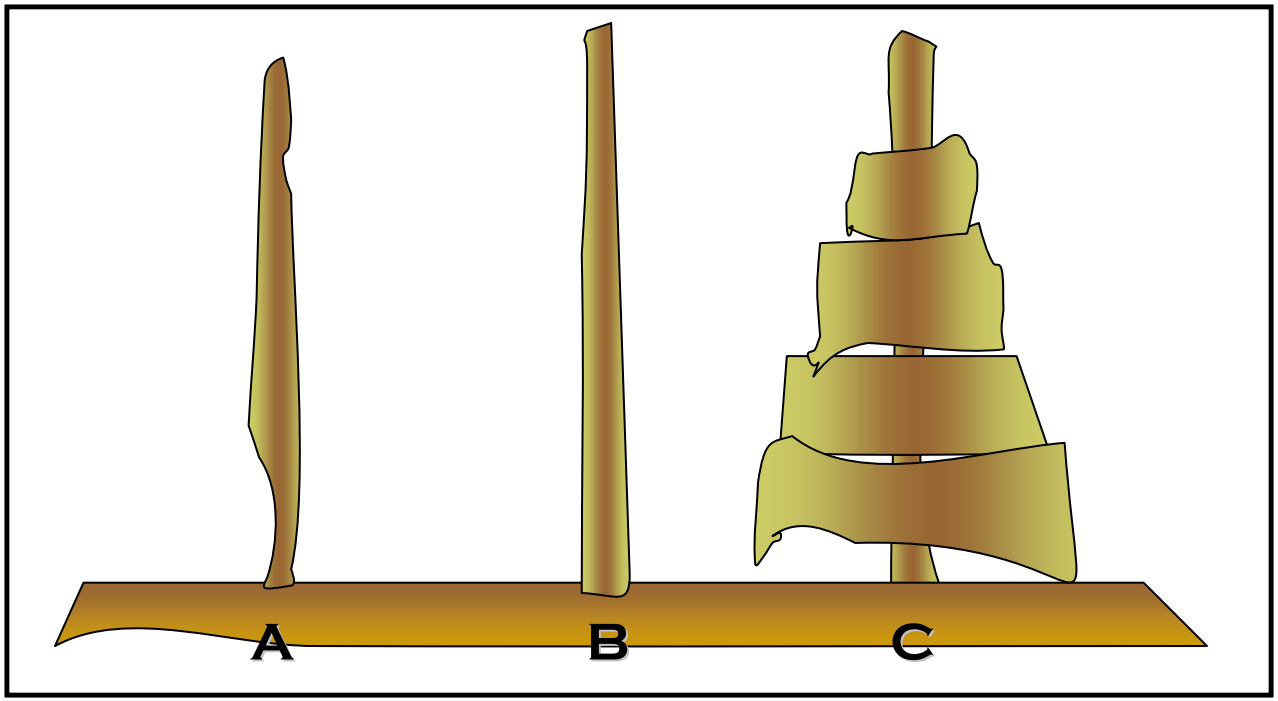
Towers of Hanoi

- Move n (4) disks from pole A to pole C
- such that a disk is never put on a smaller disk

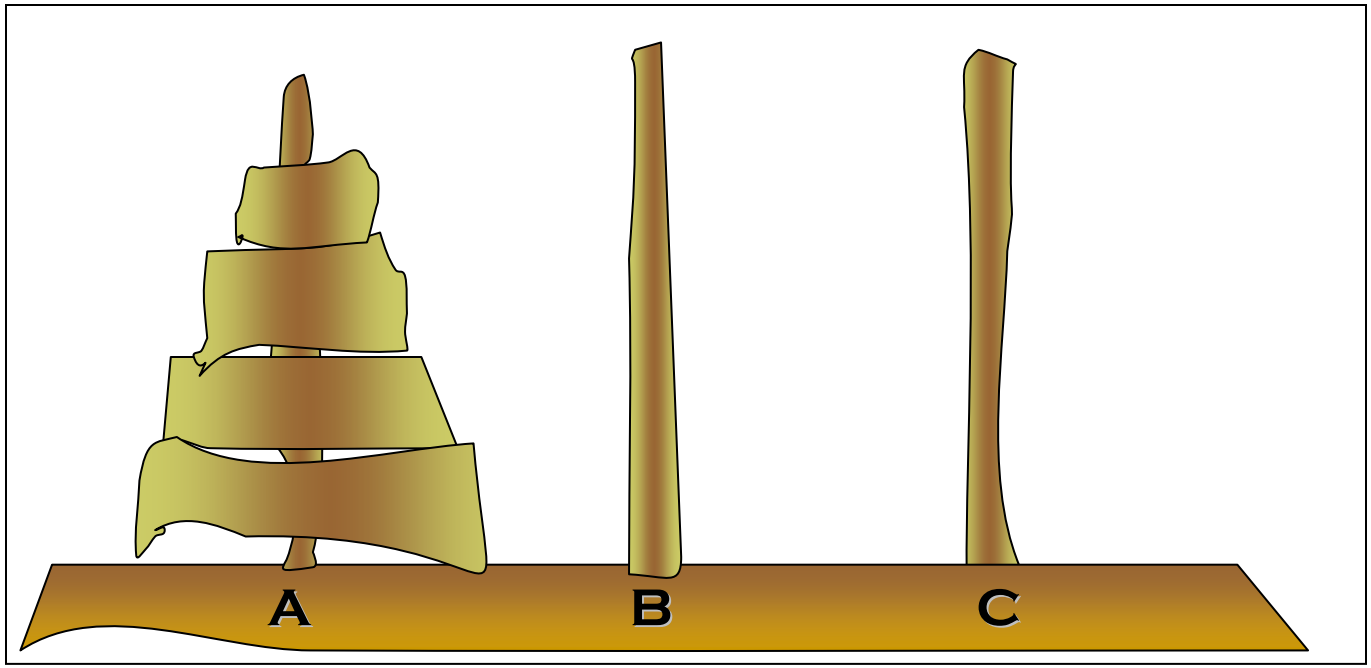


Towers of Hanoi

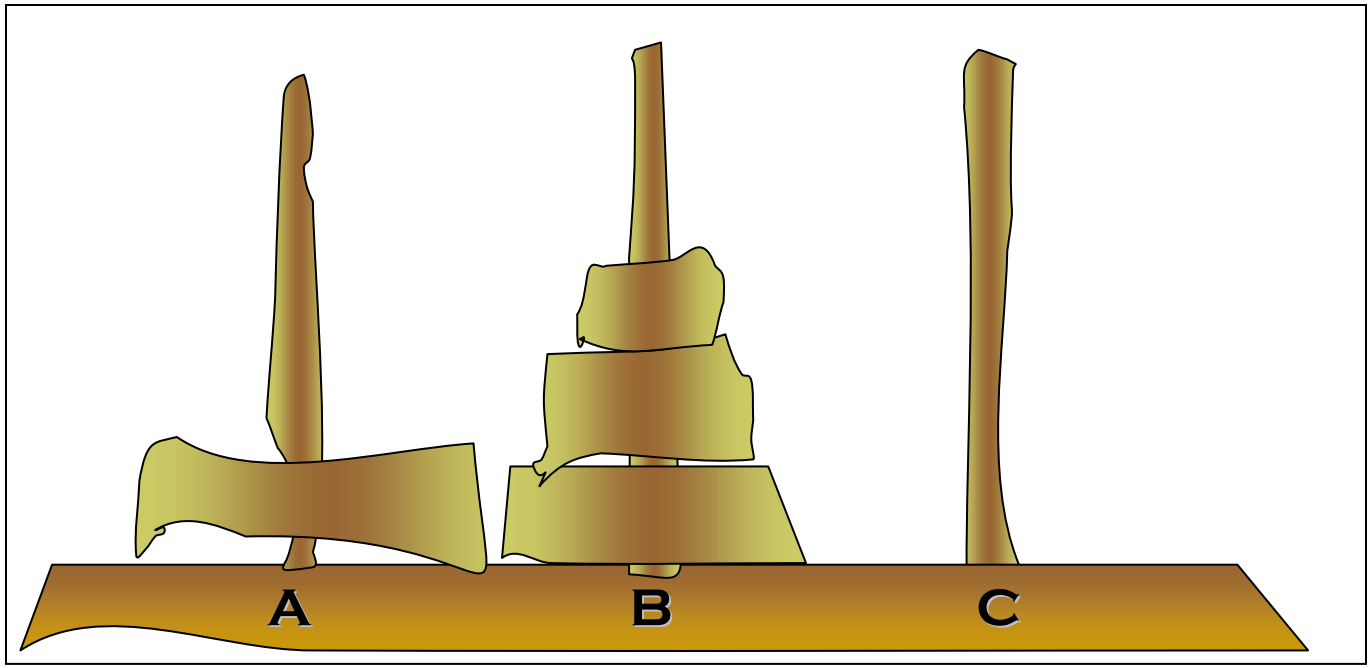
- Move n (4) disks from pole A to pole C
- such that a disk is never put on a smaller disk



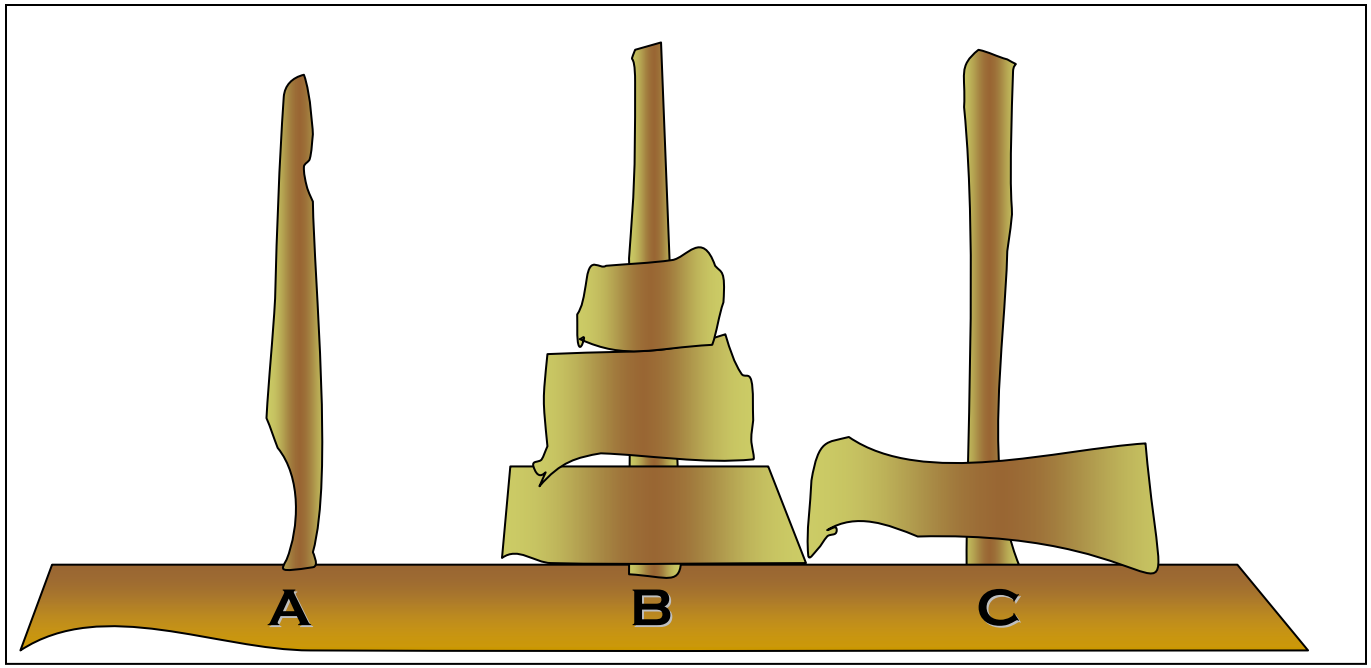
- Move n (4) disks from A to C

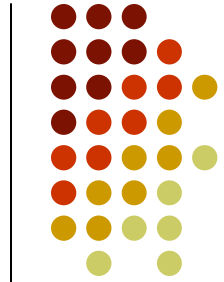


- Move n (4) disks from A to C
  - Move n-1 (3) disks from A to B



- Move n (4) disks from A to C
  - Move n-1 (3) disks from A to B
  - Move 1 disk from A to C





- Move n (4) disks from A to C
  - Move n-1 (3) disks from A to B
  - Move 1 disk from A to C
  - Move n-1 (3) disks from B to C

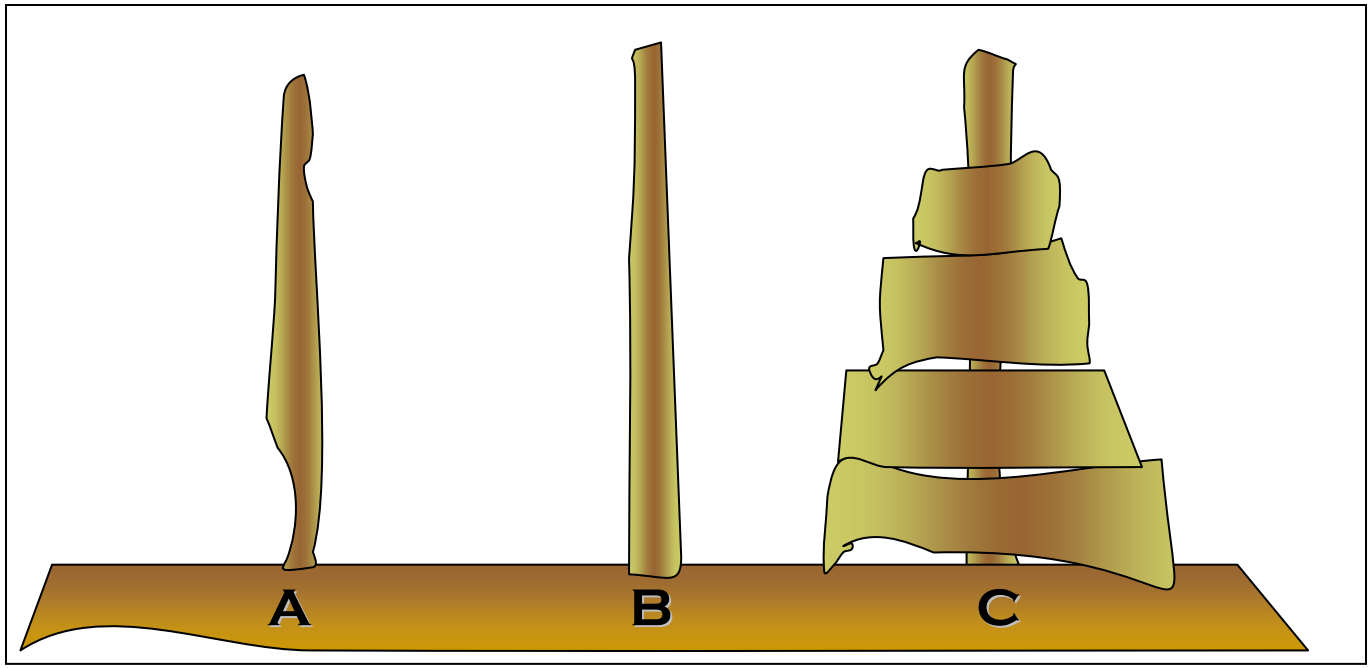


Figure 2.19a and b

a) The initial state; b) move *n* - 1 disks from *A* to *C*

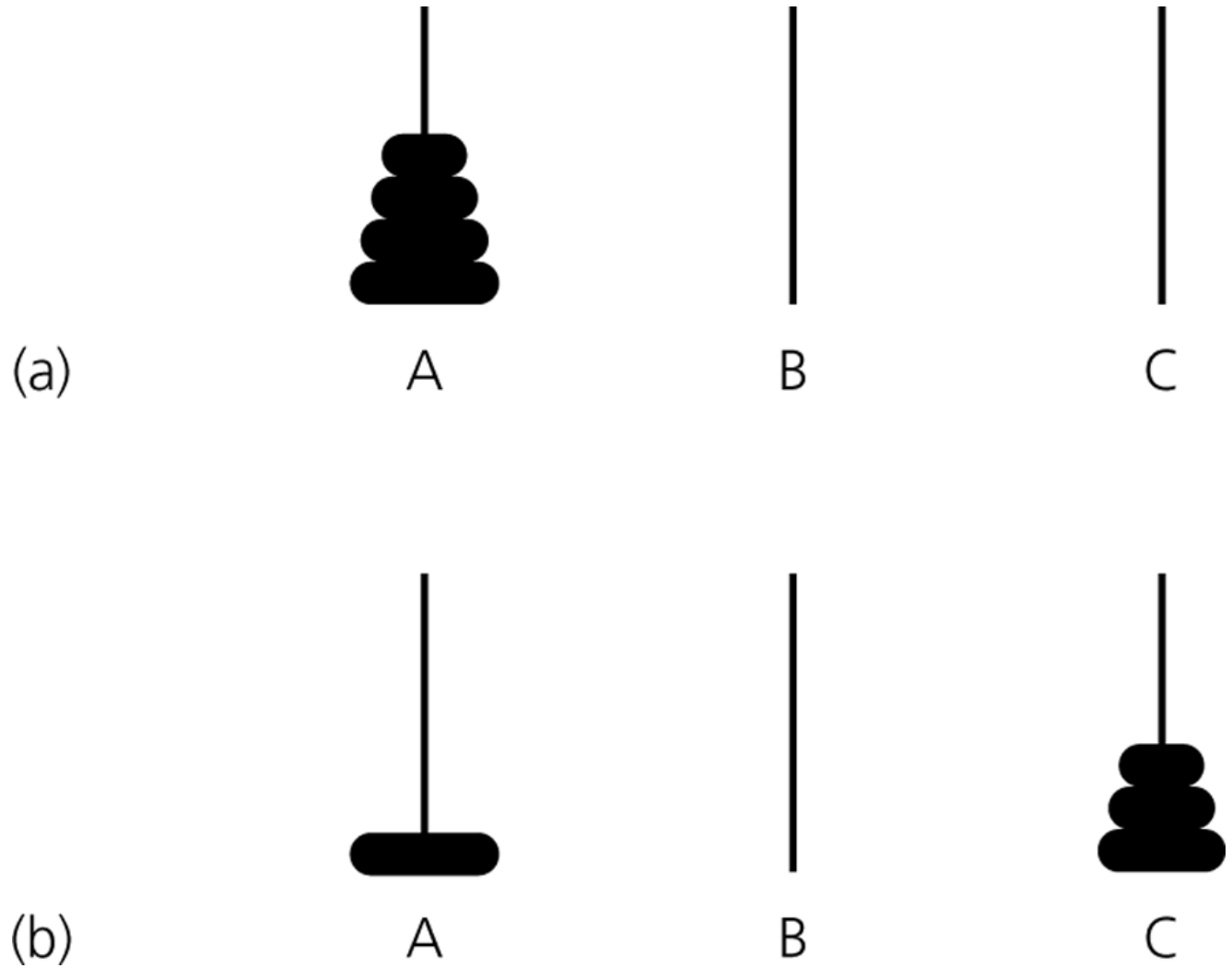
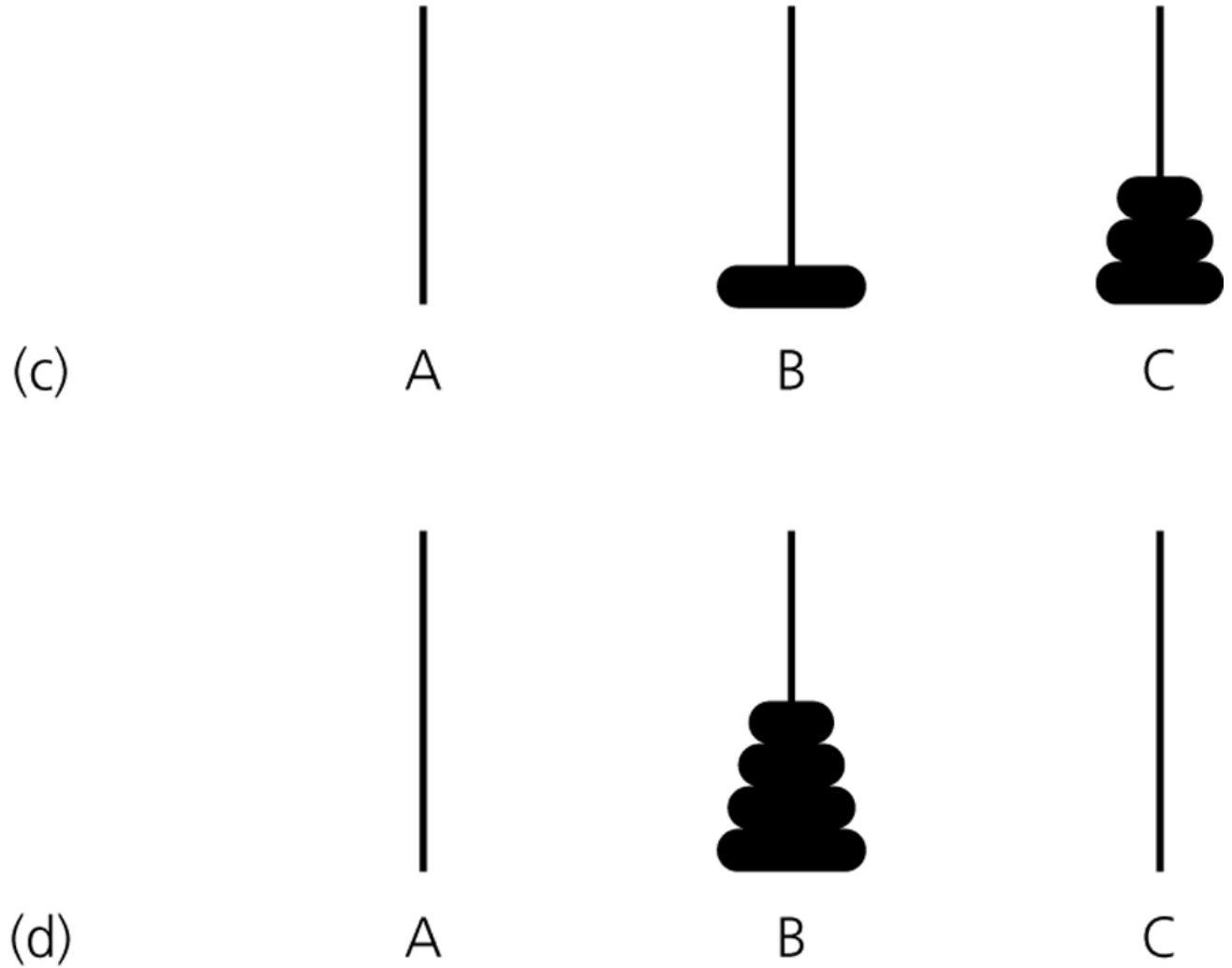
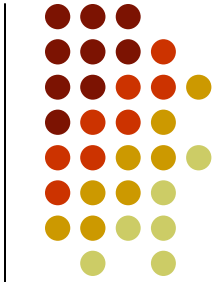


Figure 2.19c and d

c) move one disk from *A* to *B*; d) move *n* - 1 disks from *C* to *B*



Hanoi towers



```
public static void solveTowers(int count, char source,
                               char destination, char spare) {
  if (count == 1) {
    System.out.println("Move top disk from pole " + source +
                       " to pole " + destination);
  }
  else {
    solveTowers(count-1, source, spare, destination); // X
    solveTowers(1, source, destination, spare);       // Y
    solveTowers(count-1, spare, destination, source); // Z
  }  // end if
}  // end solveTowers
```

Recursion tree:

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

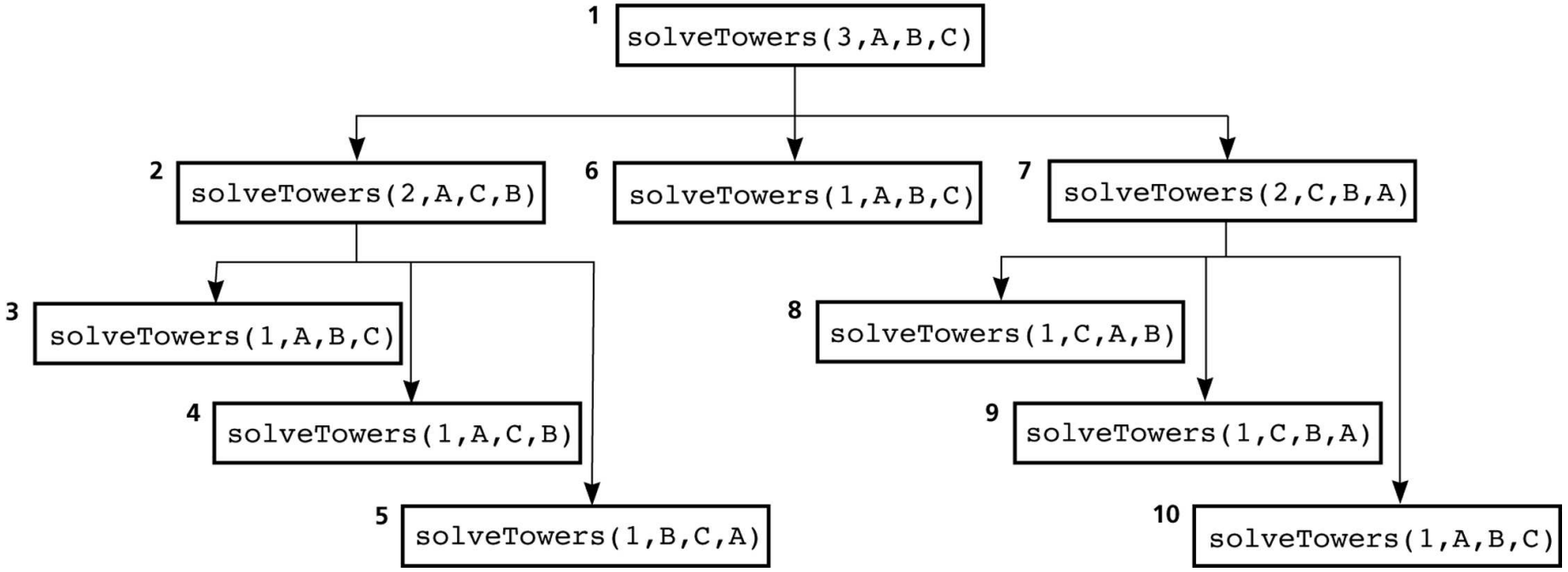
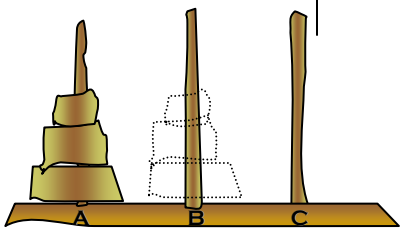


Figure 2.21a

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
```

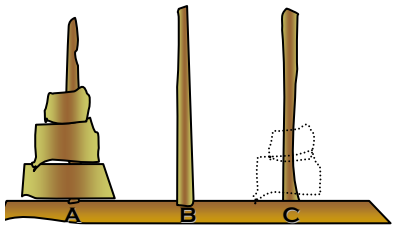


At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
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:

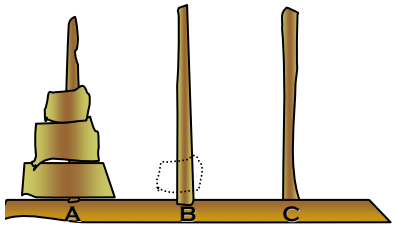
```
count  = 3
source = A
dest   = B
spare  = C
```

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
```

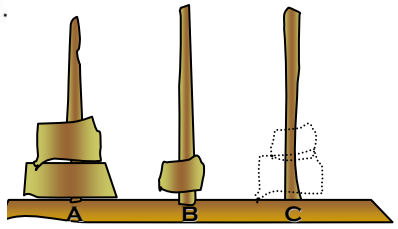
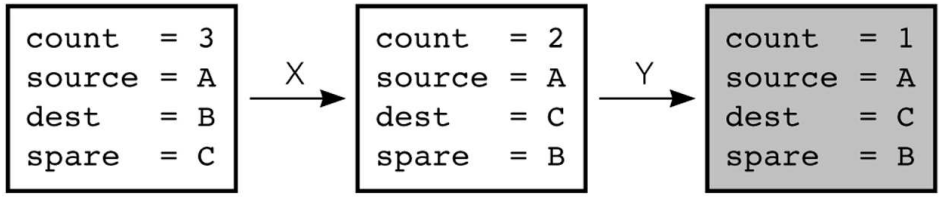
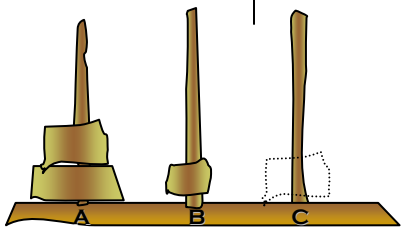


Figure 2.21b

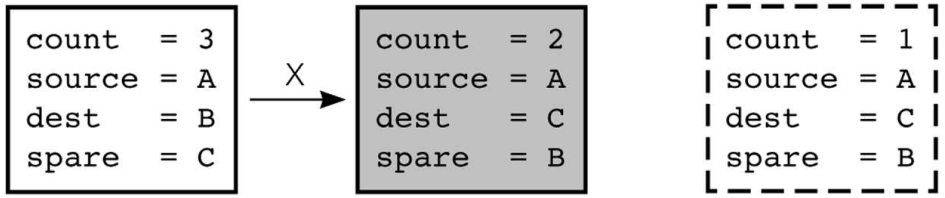
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

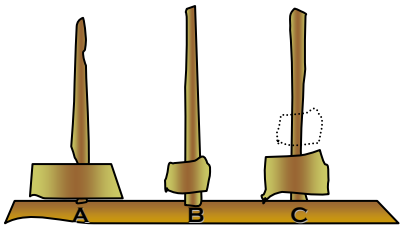
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



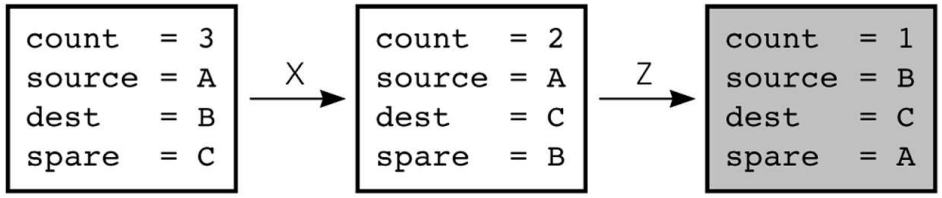


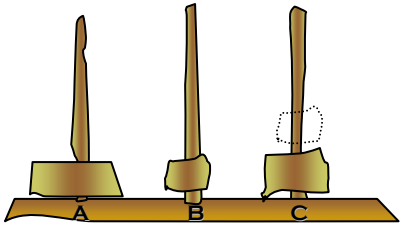
This is the base case, so a disk is moved, the return is made, and the method continues execution.



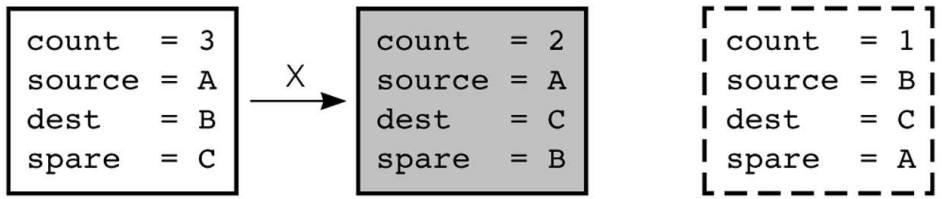


At point Z, recursive call 5 is made, and the new invocation of the method begins execution:





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



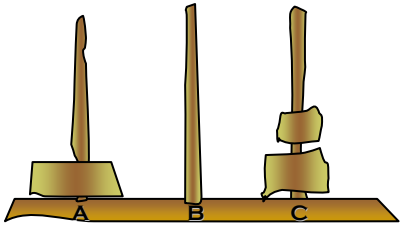
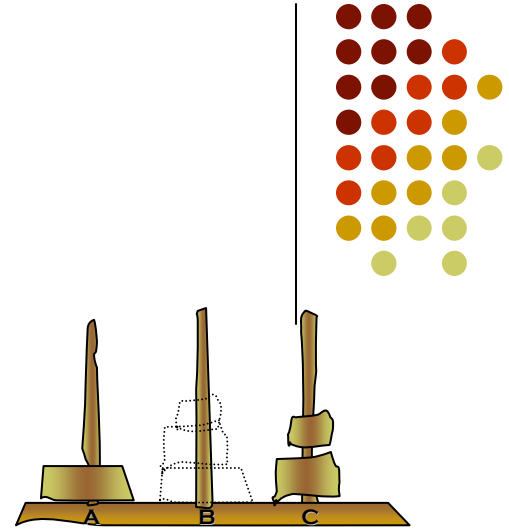


Figure 2.21c

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
```

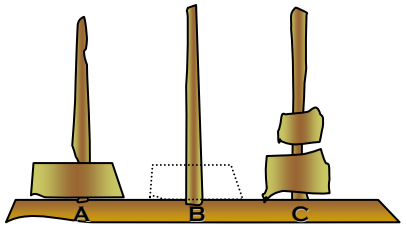
```
count  = 1
source = B
dest   = C
spare  = A
```


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 →

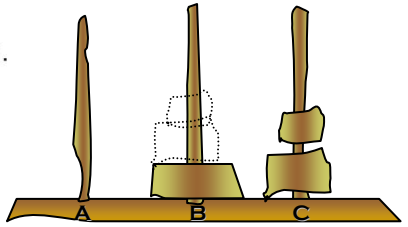
```
count  = 1
source = A
dest   = B
spare  = C
```



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
```

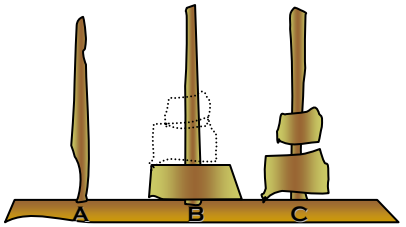
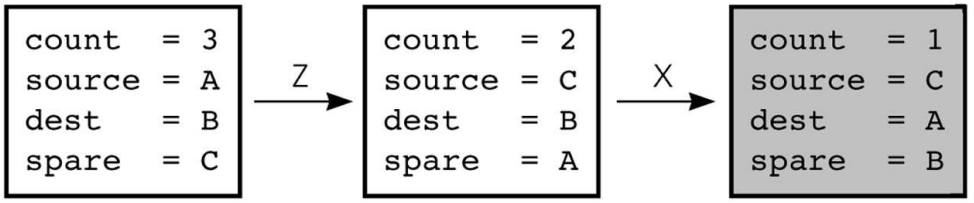
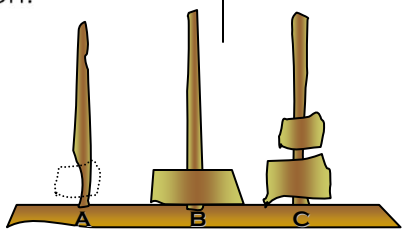


Figure 2.21d

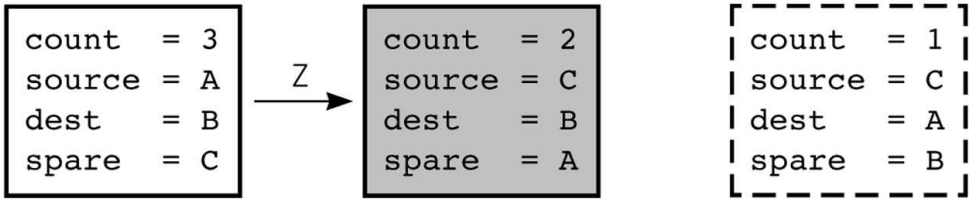
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

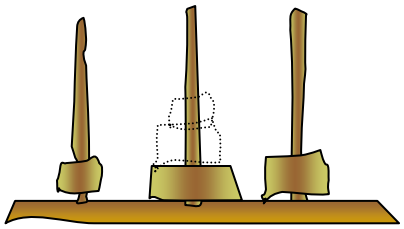
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



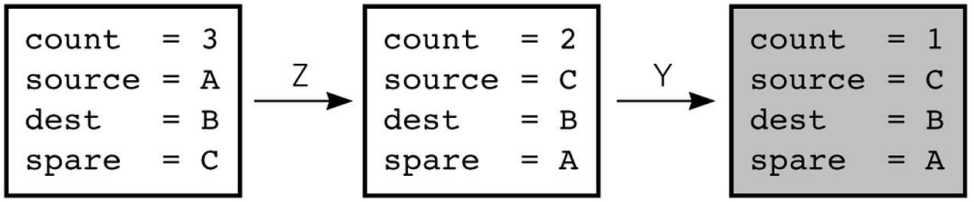


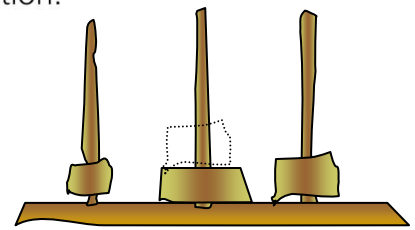
This is the base case, so a disk is moved, the return is made, and the method continues execution.



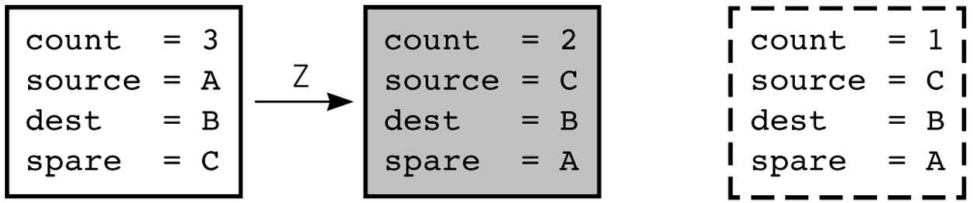


At point Y, recursive call 9 is made, and the new invocation of the method begins execution:





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



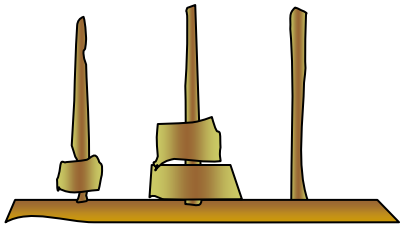
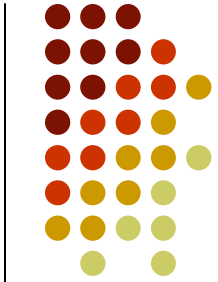
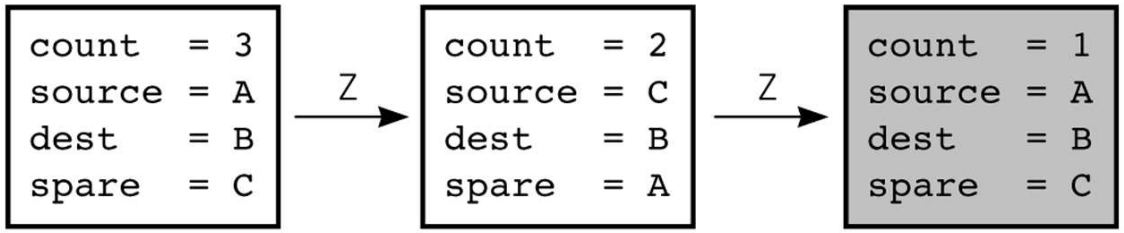


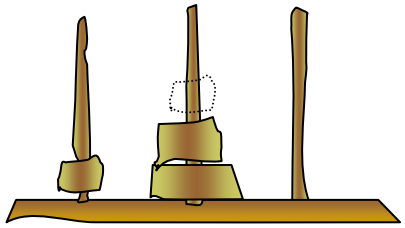
Figure 2.21e

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

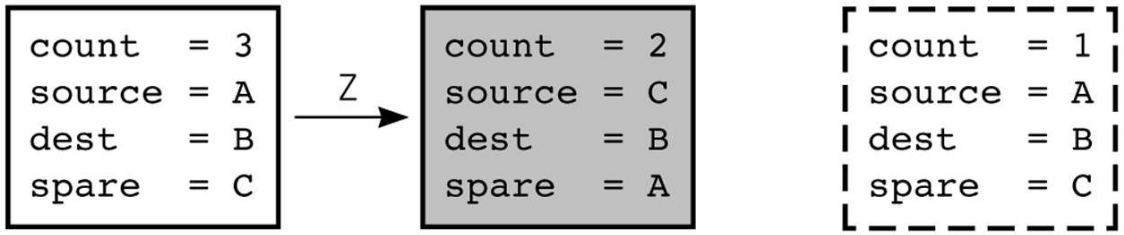


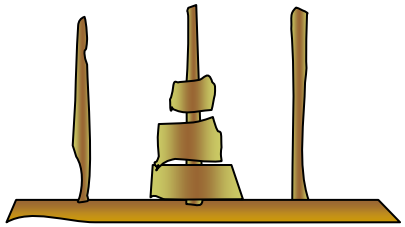
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



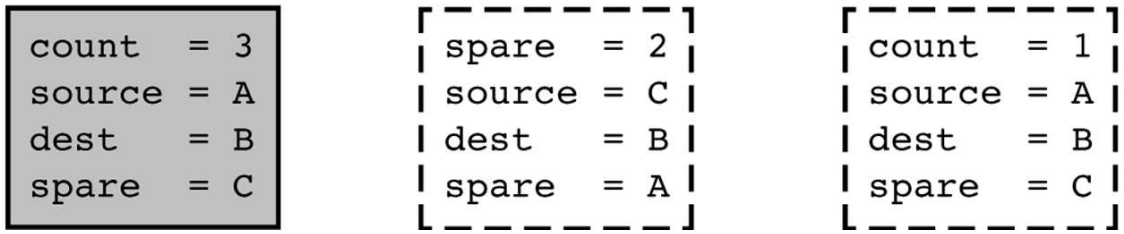


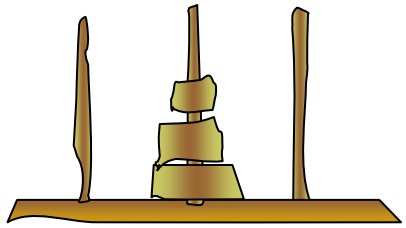
This is the base case, so a disk is moved, the return is made, and the method continues execution.



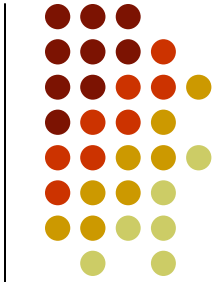


This invocation completes, the return is made, and the method continues execution.





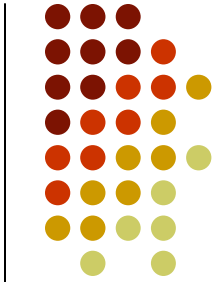
Cost of Hanoi Towers



- 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

- Guess solution and show it's correct with Mathematical Induction!

Recursive Searching



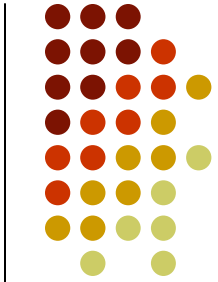
- Linear Search
- Binary Search

- Find an element in an array, return its position (index) if found, or -1 if not found.

Linear Search Algorithm (Java)

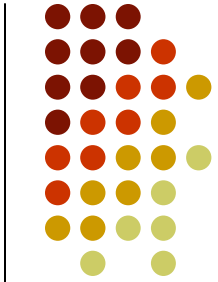
```
public int linSearch(int[] arr,
                     int target)
{
   for (int i=0; i<arr.size; i++) {
      if (target == arr[i]) {
         return i;
      }
   } //for
   return -1; //target not found
}
```

Linear Search



- Iterate through an array of *n* items searching for the target item
- The crucial instruction is equality checking (or “comparisons” for short)
  - `x.equals(arr[i]); //for objects or`
  - `x == arr[i]; //for a primitive type`
- Linear search performs at most *n* comparisons
- We can write linear search recursively

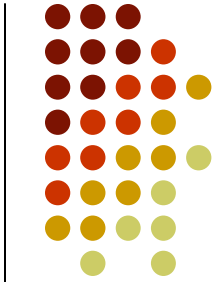
Recursive Linear Search Algorithm



- Base case
  - Found the target or
  - Reached the end of the array
- Recursive case
  - Call linear search on array from the next item to the end

```
public int recLinSearch(int[] arr,int low,int x) {
    if (low >= arr.length) { // reach the end
        return -1;
    } else if (x == arr[low]){
        return low;
    } else
        return recLinSearch(arr, low + 1, x);
    }
}
```

Binary Search Sketch



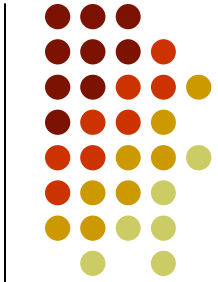
- Linear search runs in O(n) (linear) time (it requires n comparisons in the worst case)
- *If the array to be searched is sorted (from lowest to highest), we can do better:*

- Check the midpoint of the array to see if it is the item we are searching for
  - Presumably there is only a 1/n chance that it is!
    (assuming that the target is in the array)
- It the value of the item at the midpoint is less than the target then the target must be in the upper half of the array
  - So perform binary search on that half
  - and so on ….

Thinking About Binary Search

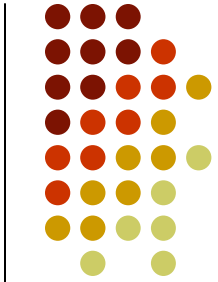
- Each sub-problem searches an array slice (or subarray)
  - So differs only in the upper and lower array indices that define the array slice
- Each sub-problem is smaller than the previous problem
  - In the case of binary search, half the size
- The final problem is so small that it is trivial
  - Binary search terminates after the problem space consists of one item or
  - When the target item is found
- Be careful when writing the terminating condition
  - When exactly do we want to stop?
    - When the search space consists of one element but
    - Only **after** that one element has been tested

Recursive Binary Search Algorithm



```
public int binSearch(
        int[] arr, int lower, int upper, int x)
{
    int mid = (lower + upper) / 2;
    if (lower > upper) {// empty interval
        return - 1; // base case
    } else if(arr[mid] == x){
        return mid; // second base case
    } else if(arr[mid] < x){
         return binSearch(arr, mid + 1, upper, x);
    } else { // arr[mid] > target
         return binSearch(arr, lower, mid - 1, x);
    }
}
```

Analyzing Binary Search

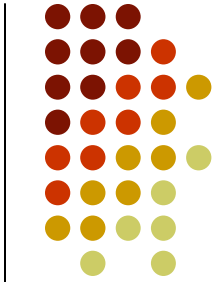


- **Best case**: 1 comparison
- **Worst case**: target is not in the array, or is the last item to be compared
  - Each recursive call halves the input size
  - Assume that $\boldsymbol{n = 2^k}$ (e.g. if n = 128, k = 7)
  - After the **first** iteration there are $\boldsymbol{n/2}$ candidates
  - After the **second** iteration there are $\boldsymbol{n/4}$ (or $\boldsymbol{n/2^2}$) candidates
  - After the **third** iteration there are $\boldsymbol{n/8}$ (or $\boldsymbol{n/2^3}$) candidates
  - After the ***k*-th** iteration there is one candidate because $\boldsymbol{n/2^k = 1}$
    - Because $\boldsymbol{n = 2^k}$, $\boldsymbol{k = \log_2 n}$
    - Thus, at most $\boldsymbol{k=\log_2 n}$ recursive calls are made in the worst case!

Binary Search vs Linear Search

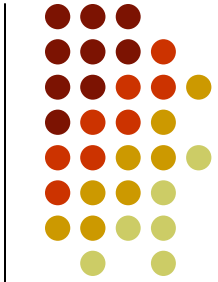
	Linear	Binary
N	N	$\log_2(N)$
10	10	4
100	100	7
1,000	1000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24

Iterative Binary Search



- Use a `while` loop instead of recursive calls
  - The initial values of lower and upper do not need to be passed to the method but
  - Can be initialized before entering the loop with lower set to 0 and upper to the length of the array-1
  - Change the lower and upper indices in each iteration
- Use the (negation of the) base case condition as the condition for the loop in the iterative version.
  - Return a negative result if the while loop terminates without finding the target

Binary Search Algorithm (Java)

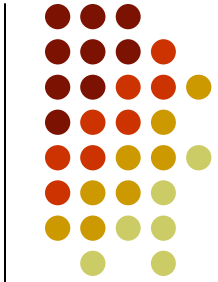


```
public int binSearch(int[] arr, int target){
    int lower = 0;
    int upper = arr.length - 1;

    while (lower <= upper){
        int mid = (lower + upper) / 2;
        if (target == arr[mid]) {
            return mid;
        } else if (target > arr[mid]) {
            lower = mid + 1;
        } else { //target < arr[mid]
            upper = mid - 1;
        }
    } //while
    return -1; //target not found
}
```

Index of the first and last elements in the array

Recursion Disadvantage 1

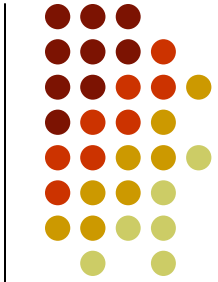


- Recursive algorithms have more overhead than similar iterative algorithms
  - Because of the repeated method calls (storing and removing data from call stack)
  - This may also cause a “stack overflow” when the call stack gets full

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

    (Especially, when computation continues after the recursive call -> we often need to remember value of some local variable -> stacks can be often used to store that information.)

Recursion Disadvantage 2



- 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
- Even if the solution was determined using recursion such algorithms should be implemented iteratively

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

Function Analysis for call `fib(5)`

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