Algorithms (CS-204)

Recursion

What is recursion?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first

- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases} \qquad \text{(}\textit{recursive}\text{ solution)}$$

$$n! = \begin{cases} \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases} \qquad \text{(}\textit{closed form}\text{ solution)}$$

Recursion vs. iteration

- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

Three Things to remember

1. Base Condition:

   Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?

2. Progress towards Base condition and eventually meet base condition:

   Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

3. Correctness:

   Assuming that the recursive call(s) work correctly, does the whole function work correctly?

How is recursion implemented?

- What happens when a function gets called?

```
int f1(int x)
{
 return 2*x;
}


int f2(int x)
{
 int z,y;
 ……………… // other statements
 z = f1(x) + y;

 return z;
}

int main(){
int x=5;
print f2(5);
return 0;
}
```

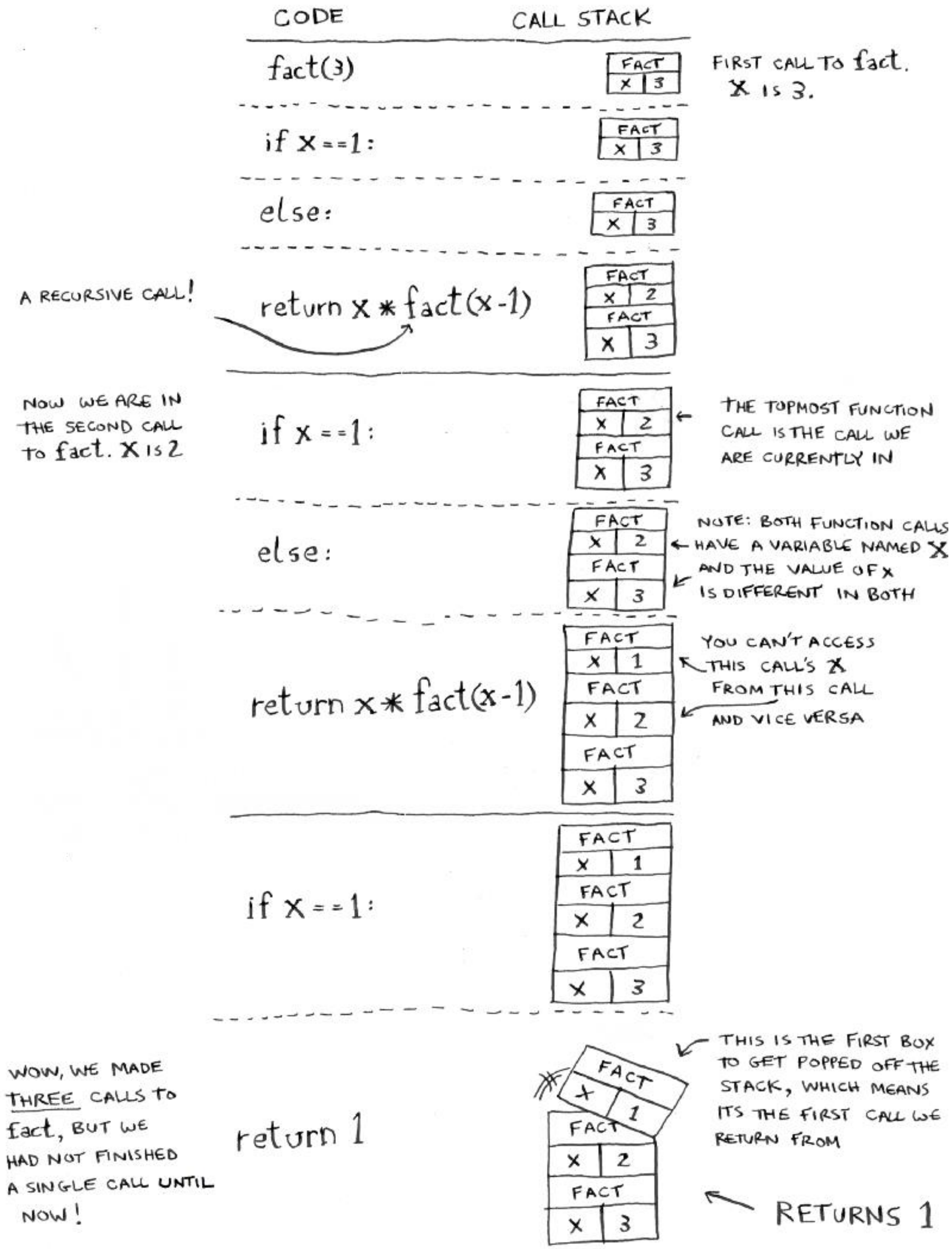
What happens when a function is called?

- An **activation** record is stored into a stack **(run-time stack)**
  1) The computer has to stop executing function **f2** and starts executing function **f1**
  2) Since it needs to come back to function **f2** later, it needs to store everything about function **f2** that is going to need (**x, y, z**, and the place to start executing upon return)
  3) Then, **x** from **f2** is bounded to **x** of **f1**
  4) Control is transferred to function **f1**

Factorial

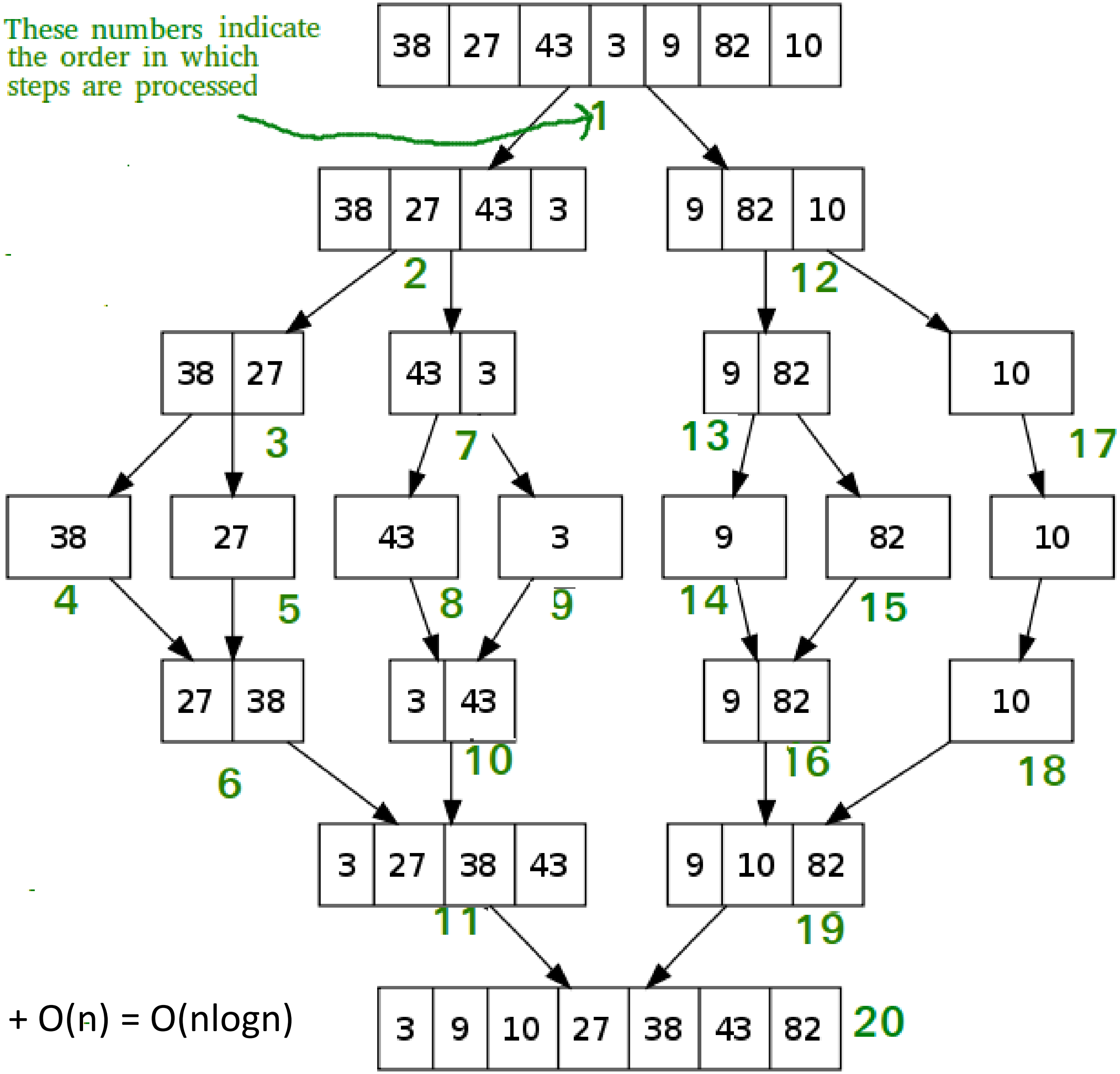
- function fact(x)

```
{
	if (x == 1) {
		return 1;
	}
	else {
		return x * fact(x-1);
	}
}
```


CODE
CALL STACK
fact(3)
FIRST CALL TO fact. X IS 3.
if x==1:
else:
return x * fact(x-1)
A RECURSIVE CALL!
NOW WE ARE IN THE SECOND CALL TO fact. X IS 2
if x==1:
THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN
else:
NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF X IS DIFFERENT IN BOTH
return x * fact(x-1)
YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA
if x==1:
return 1
WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!
THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS ITS THE FIRST CALL WE RETURN FROM
RETURNS 1
FACT
X
1
2
3

Few More Examples

Merge Sort



$T(n) = 2T(n/2) + O(n) = O(n\log n)$

Merge Sort

- **Input: Given an unordered list of integers $L=[l_1,l_2,l_3 \ldots l_n]$ where li $\in$ Integers,**
- **Output $L1= [m_1,m_2,\ldots..m_n]$ and $m_i <= m_{i+1}$ for all $1<i<n-1$ and $m_i==l_j$ for $1<i,j<=n$ and set(L)==set(L1)**

- **MergeSort(arr[], l, r)**

If r > l

1. Find the middle point to divide the array into two halves: middle m = (l+r)/2

2. Call mergeSort for first half: Call mergeSort(arr, l, m)

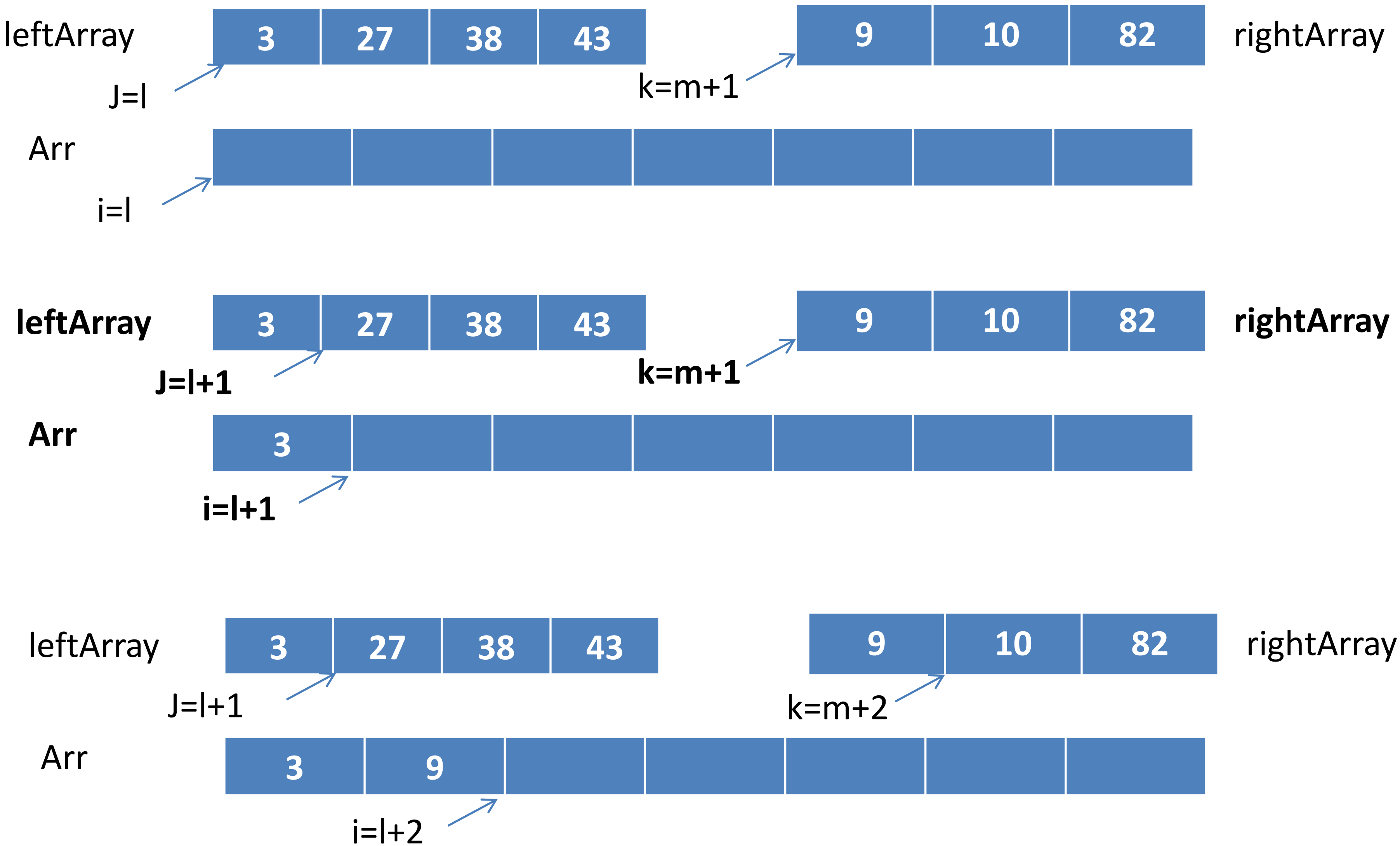
3. Call mergeSort for second half: Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3: Call merge(arr, l, m, r)

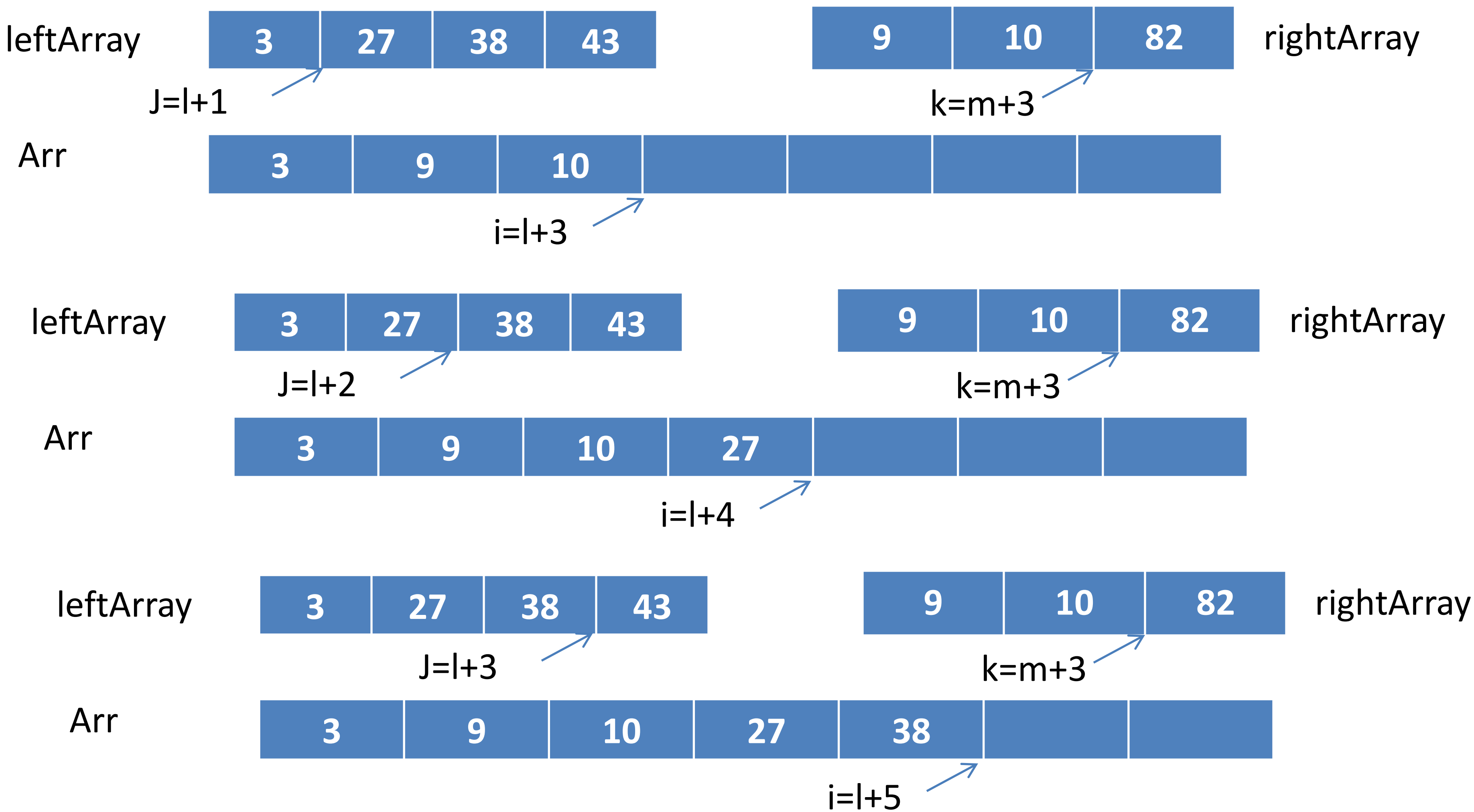
```
merge(arr, l, m, r) {
  int leftArray[m-l+1],rightArray[r-m];
  for i:l to m
    – leftArray[i-l]=arr[i]
  for i:m+1 to r
    – rightArray[i-m-1]=arr[i]
  i=l,j=l,k=m+1;
  While(j<=m and k<=r)
    – If(leftArray[j]<=rightArray[k])
        • Arr[i++]=leftArray[j++];
```

- else
  - Arr[i++]=rightArray[k++];
- While(i<=r)
  - if(j<=m)
    - Arr[i++]=leftArray[j++]
  - else
    - Arr[i++]=rightArray[k++]

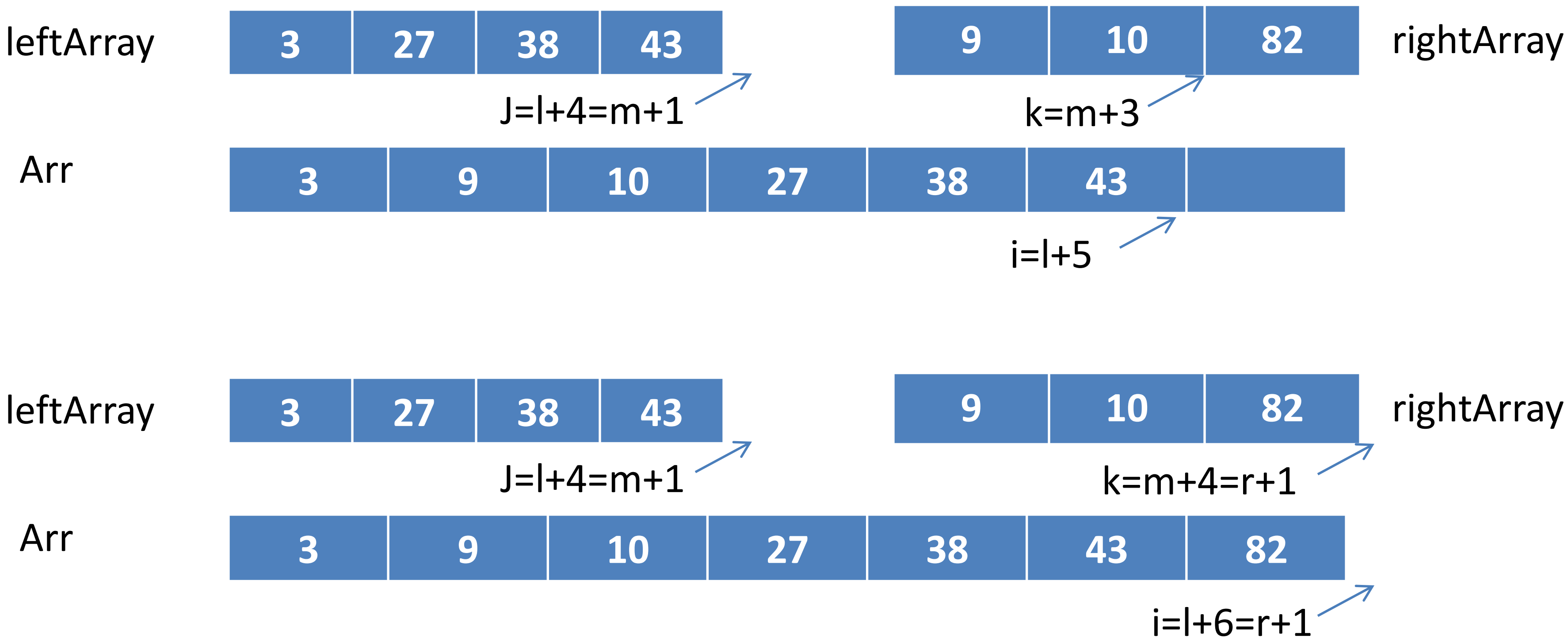
Merge Example



Merge Example



Merge Example



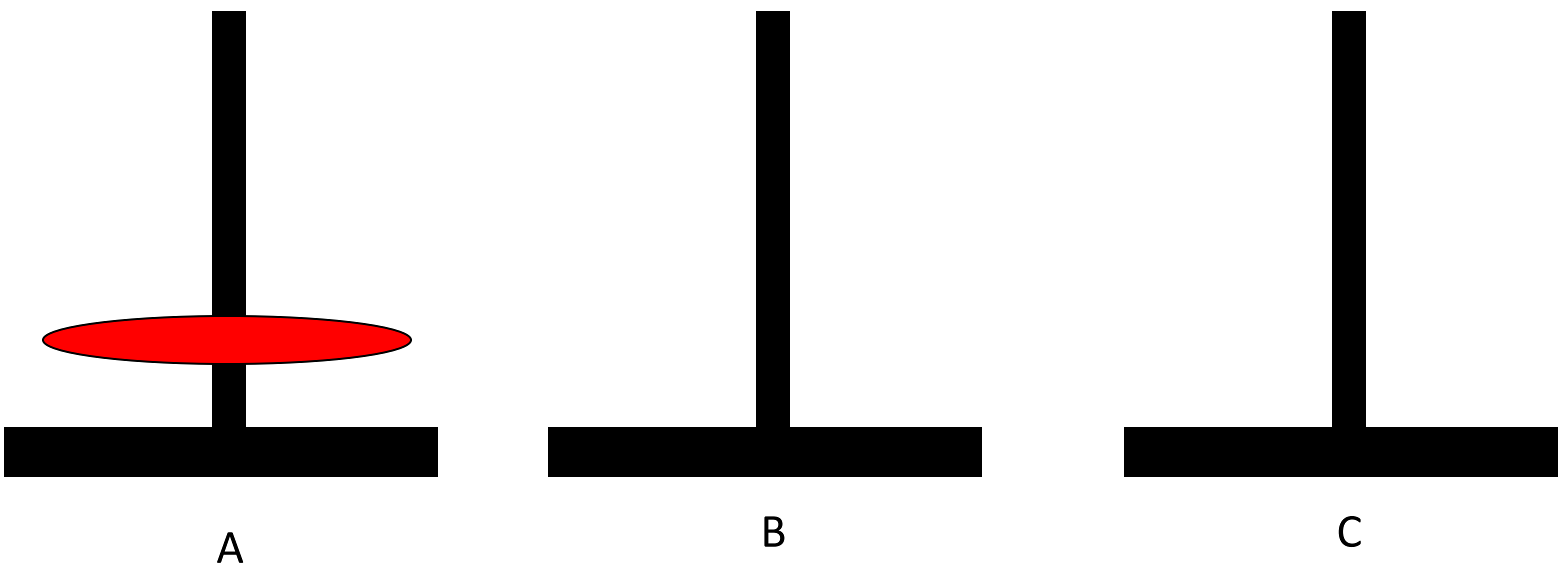
Homework

- Try to develop a recursive version for merging two sorted arrays.

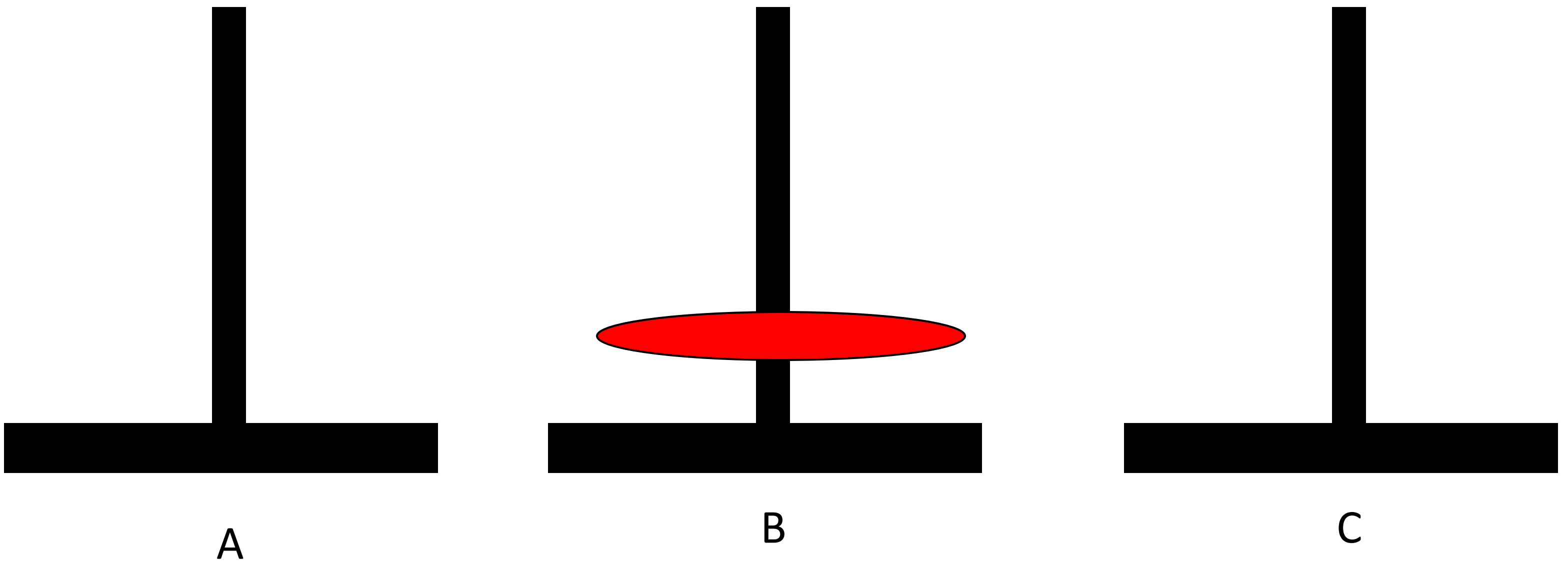
Tower of Hanoi

- There are three towers A, B, C
- There are n disks, with decreasing sizes (largest disk is placed at the bottom of the tower and on the top smallest size disk is placed), on the first tower A
- You need to move all of the disks from A to B following few rules
  - You can move only one disk at a time
  - You can move only that disk which is currently at the top of any stack
  - Larger disks can not be placed on top of a smaller disk
  - Tower C can be used to temporarily hold disks

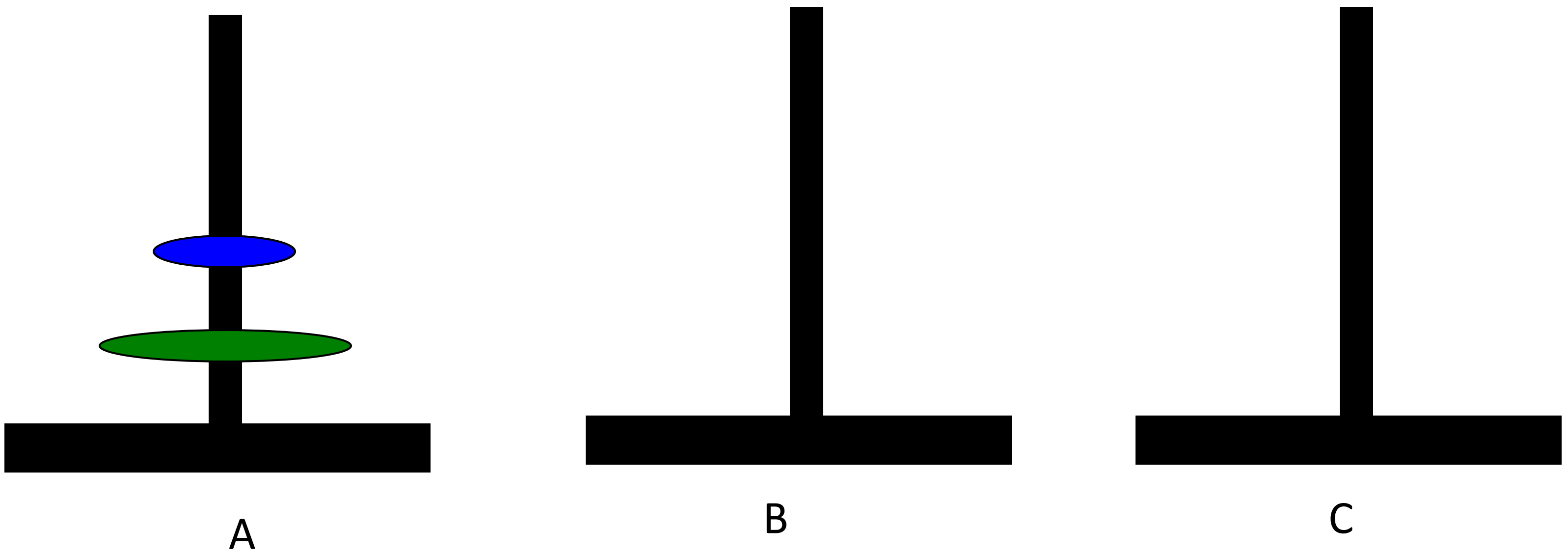
Base Case

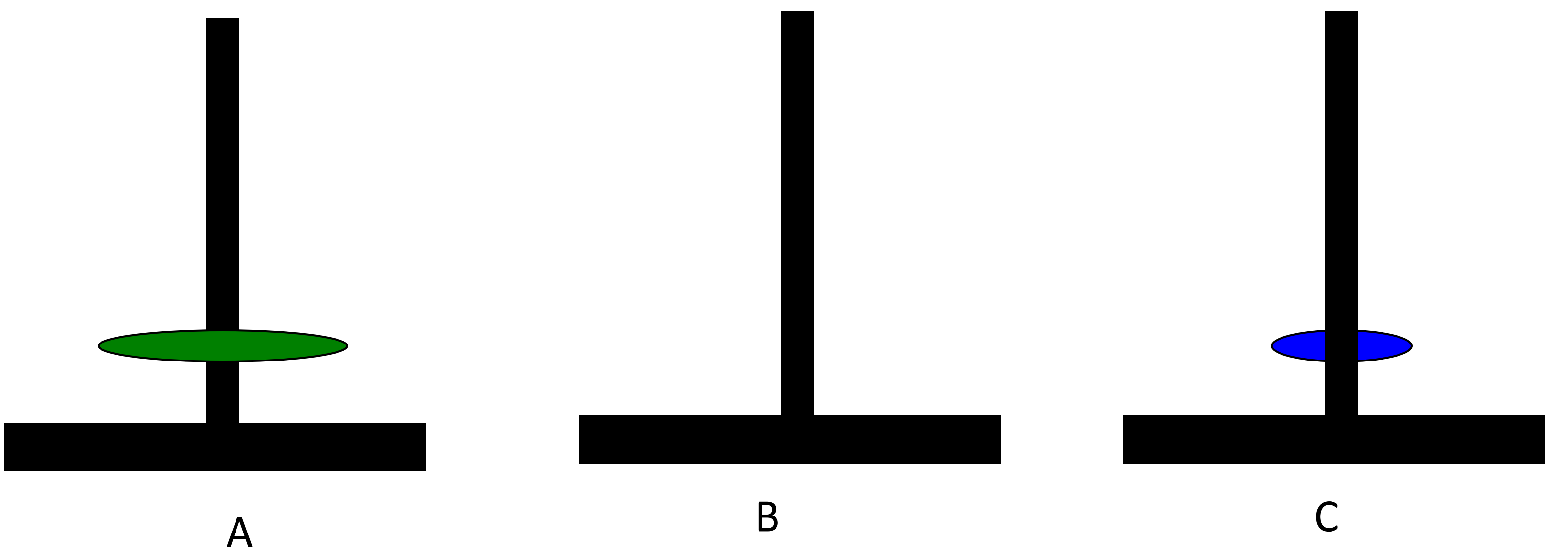


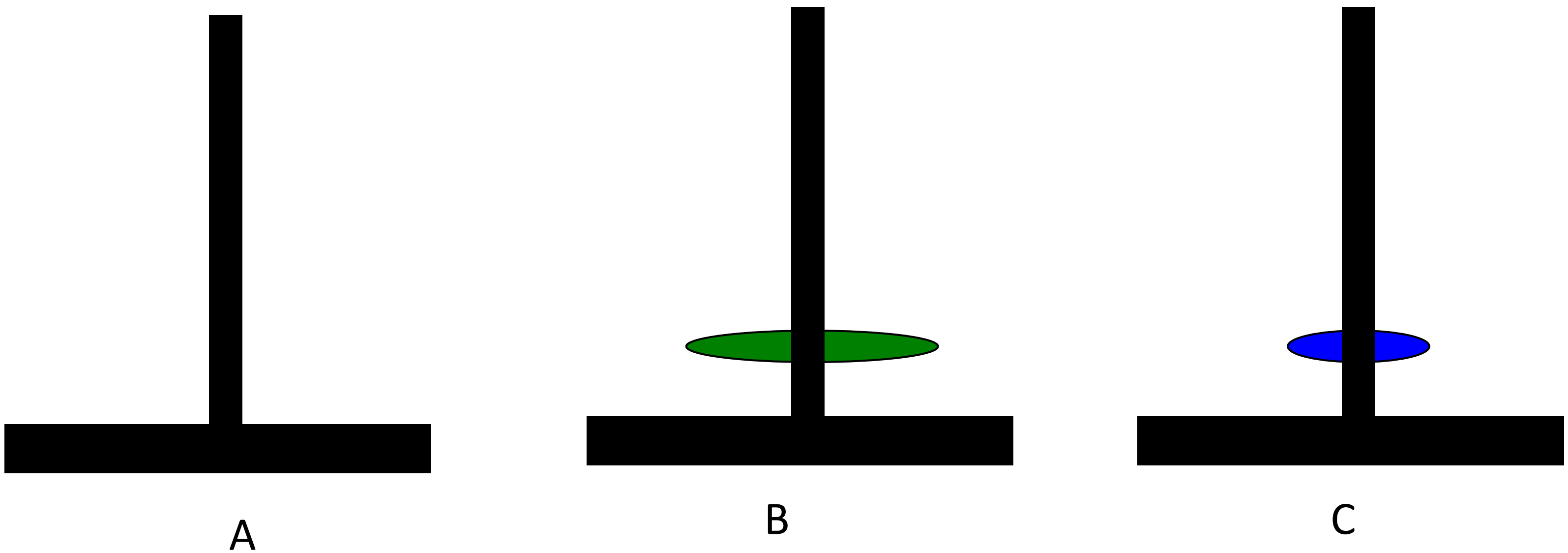
Base Case

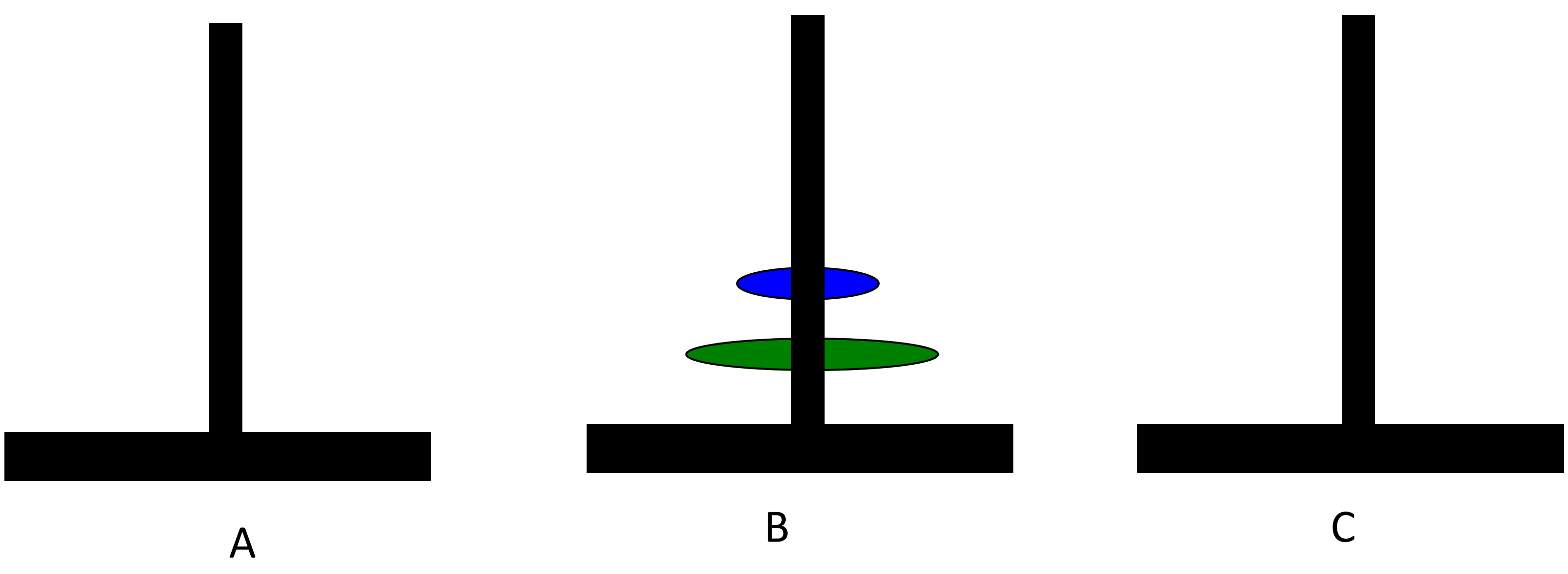


Using Base case to solve problem with 2 disks

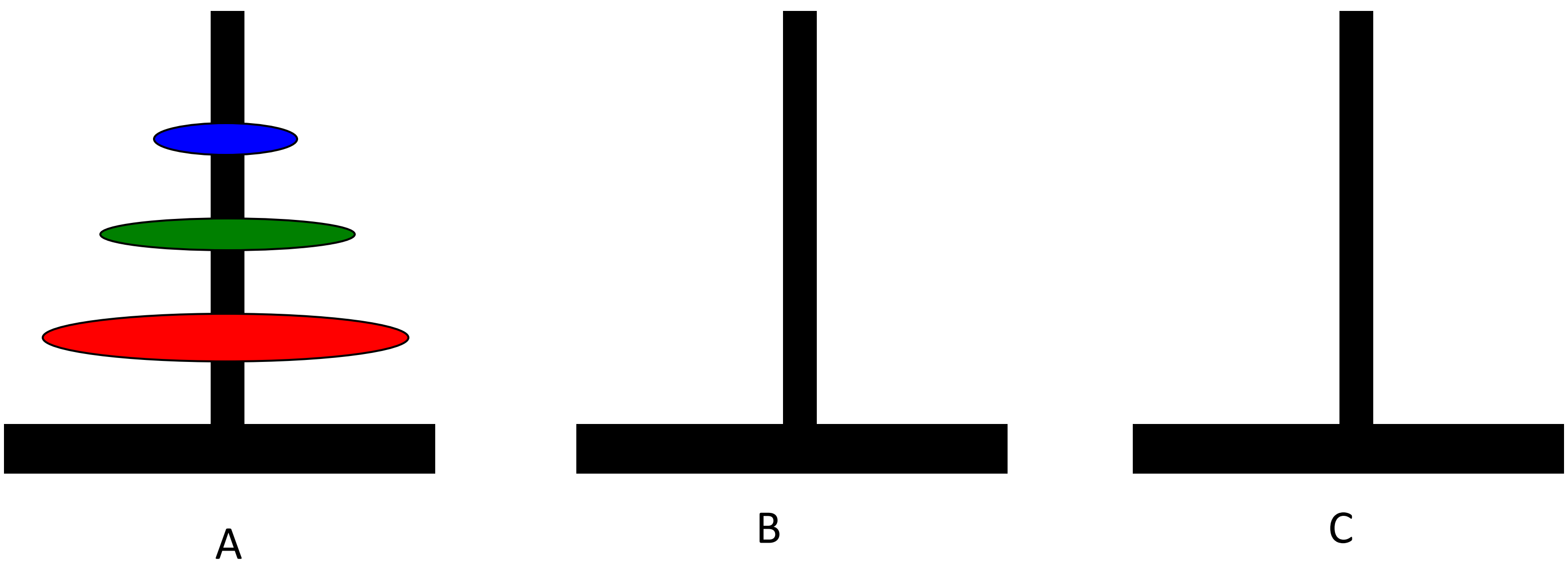



A
B
C

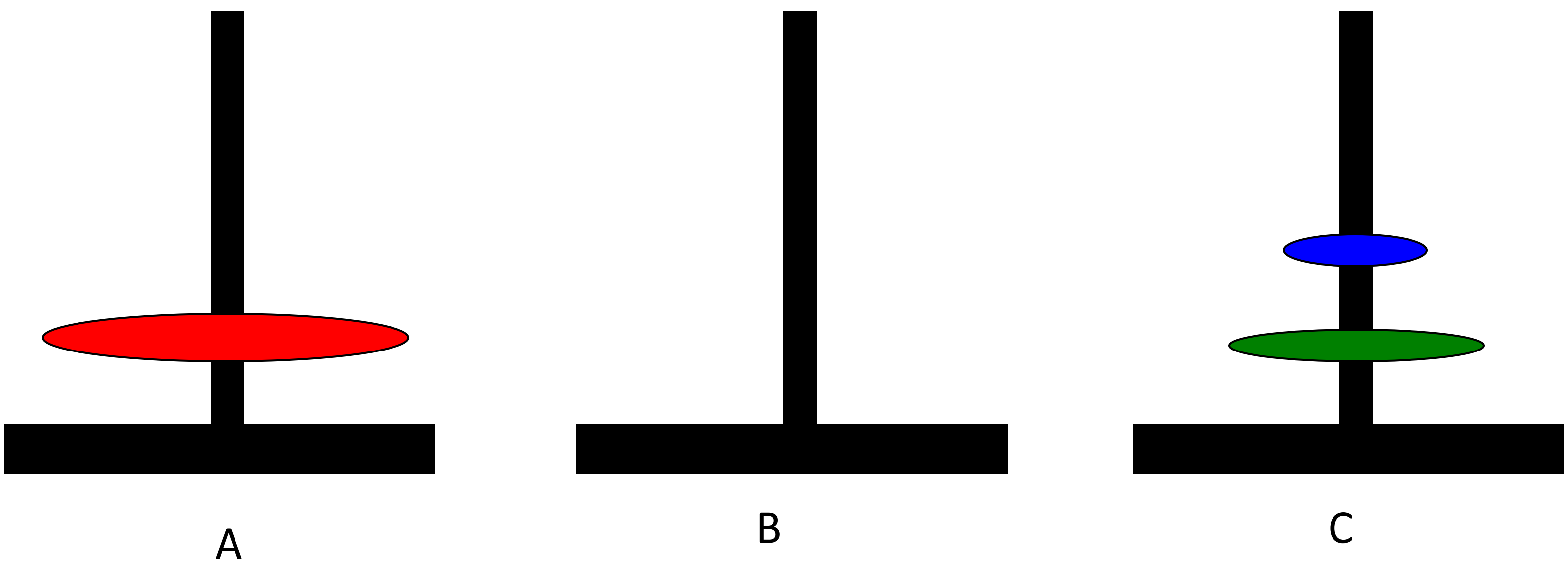

A
B
C


A
B
C

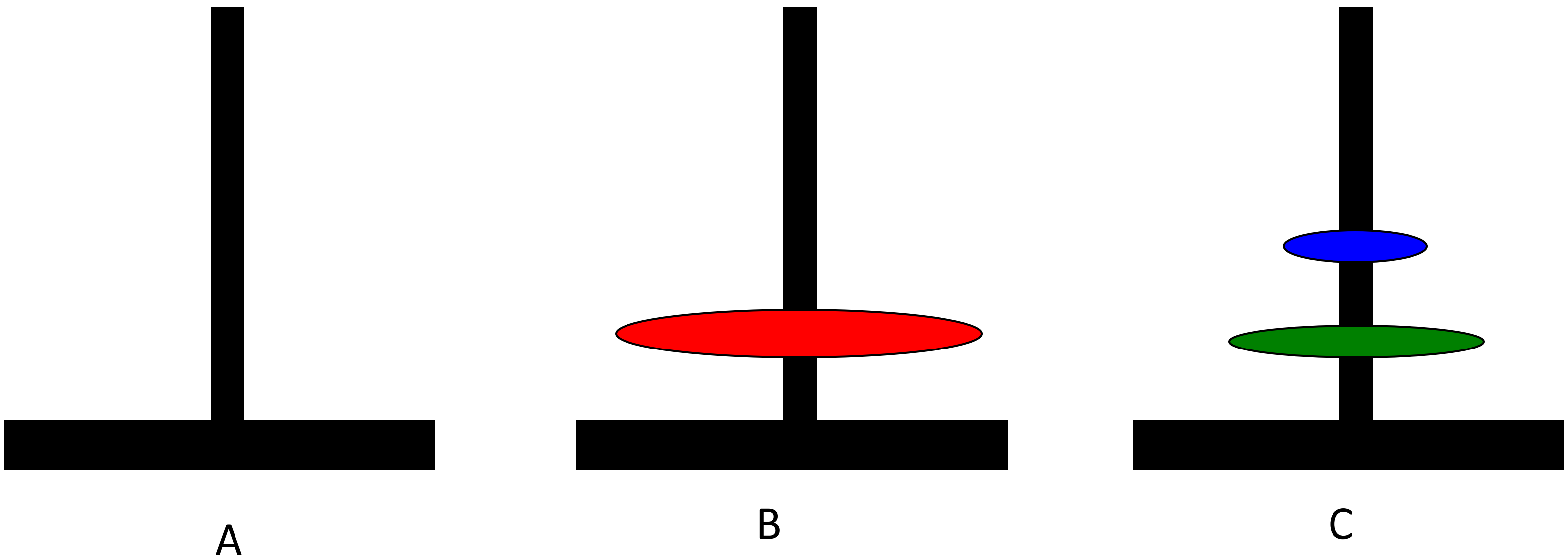
Using solution of 2 disks to solve problem with 3 disks



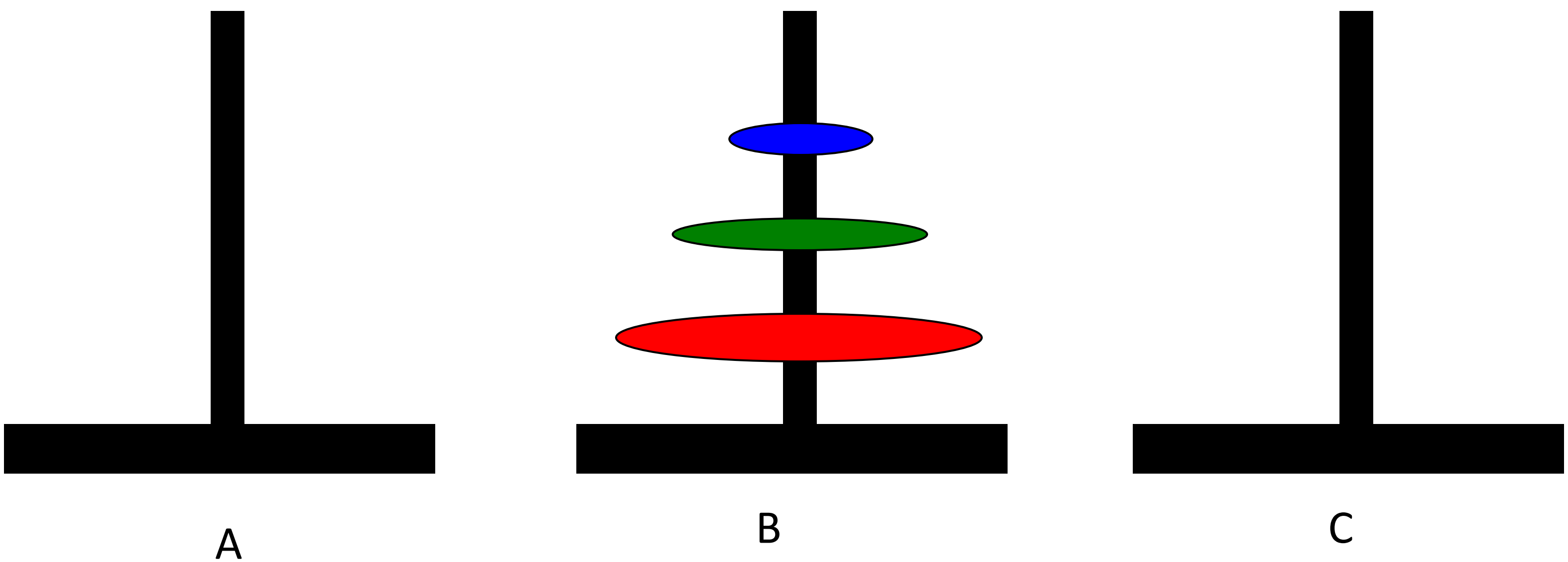
Recursive Solution



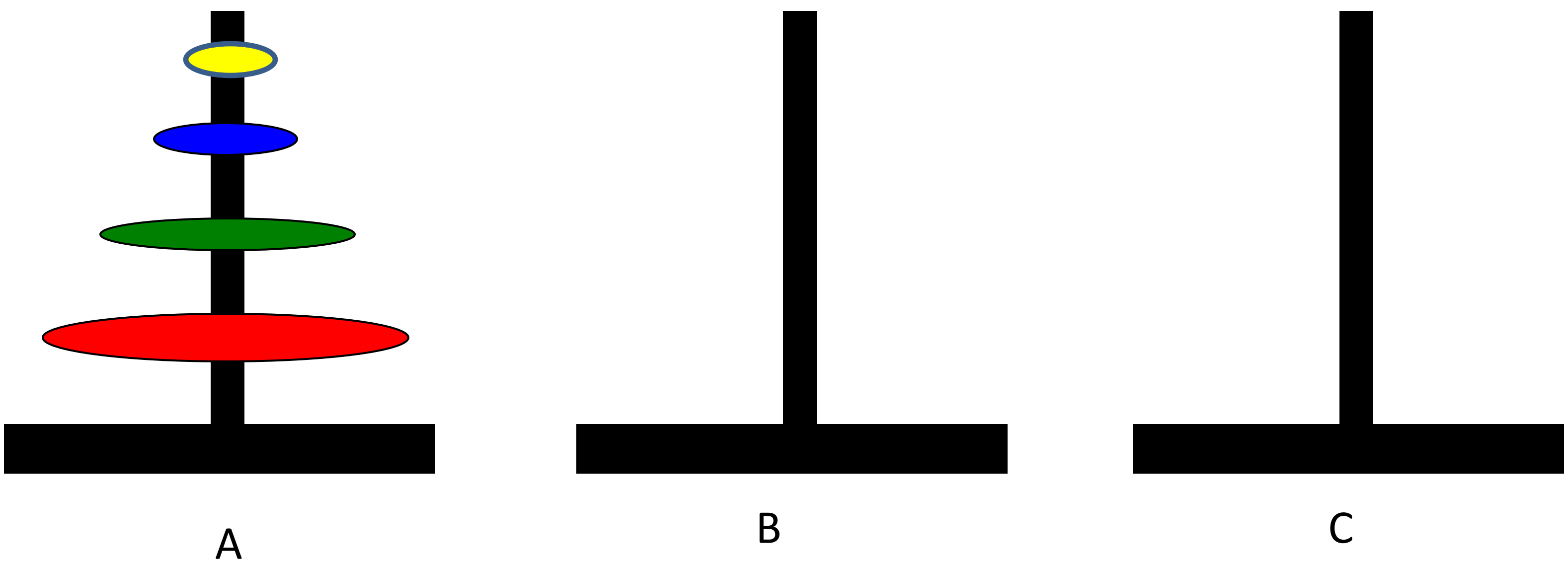
Recursive Solution



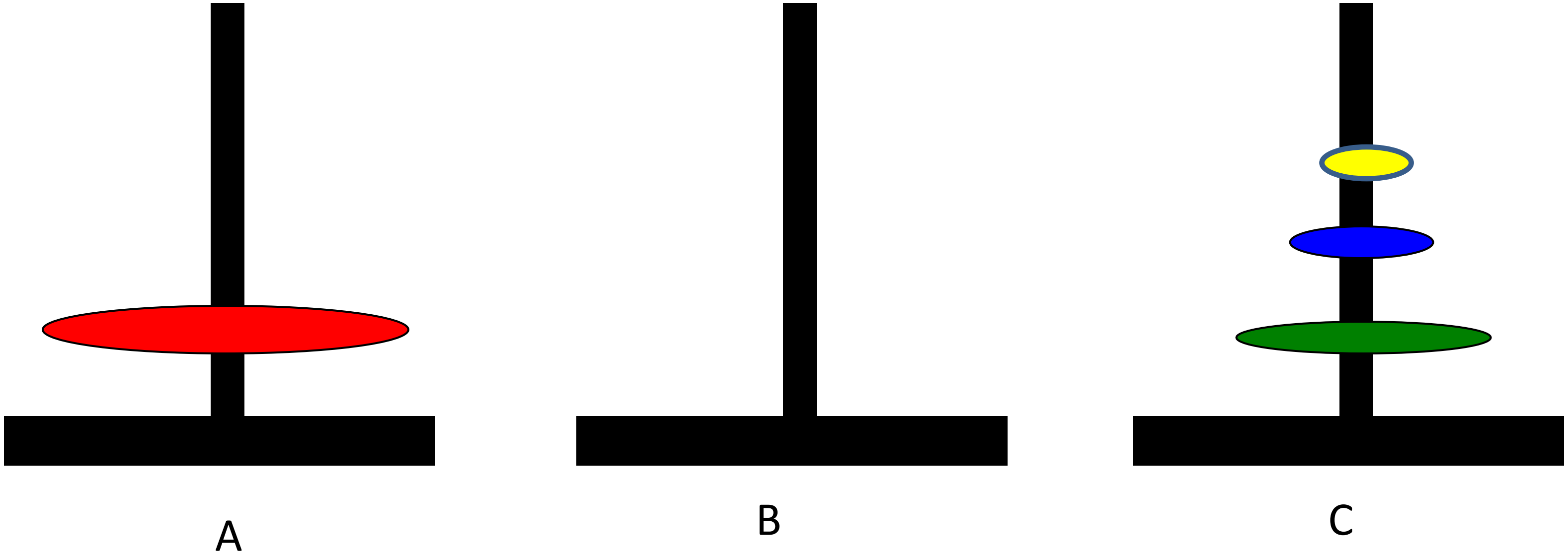
Recursive Solution



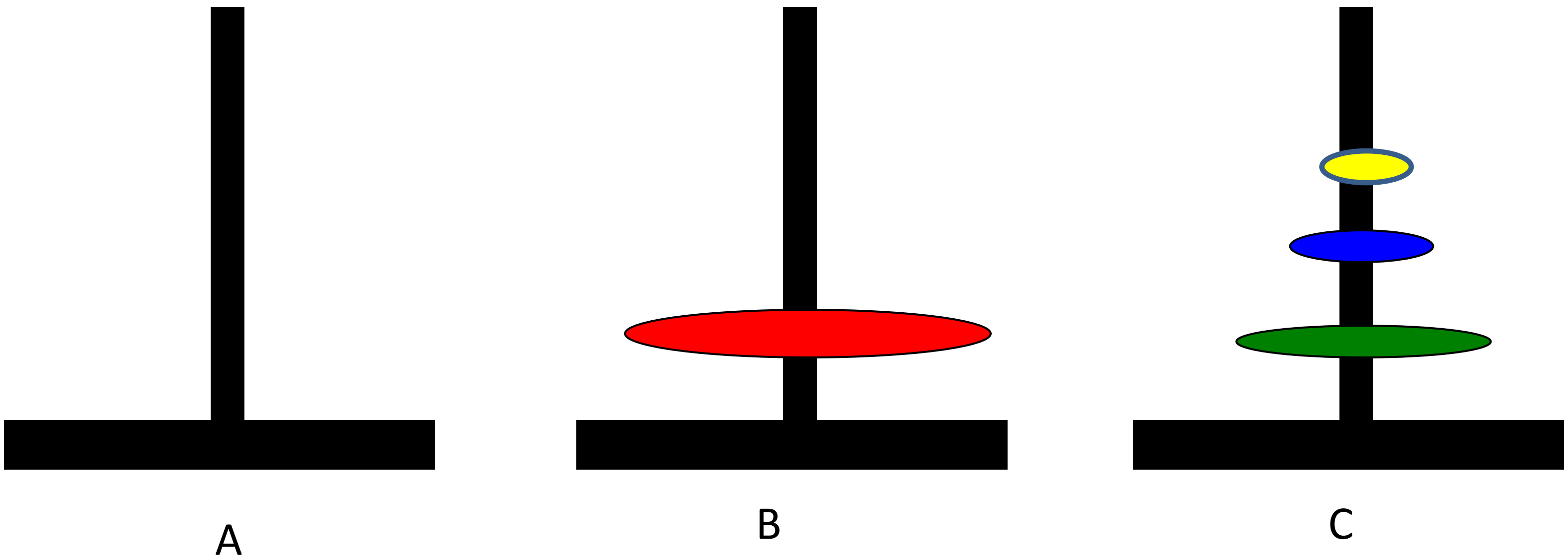
Using solution of 3 disks to solve problem with 4 disks



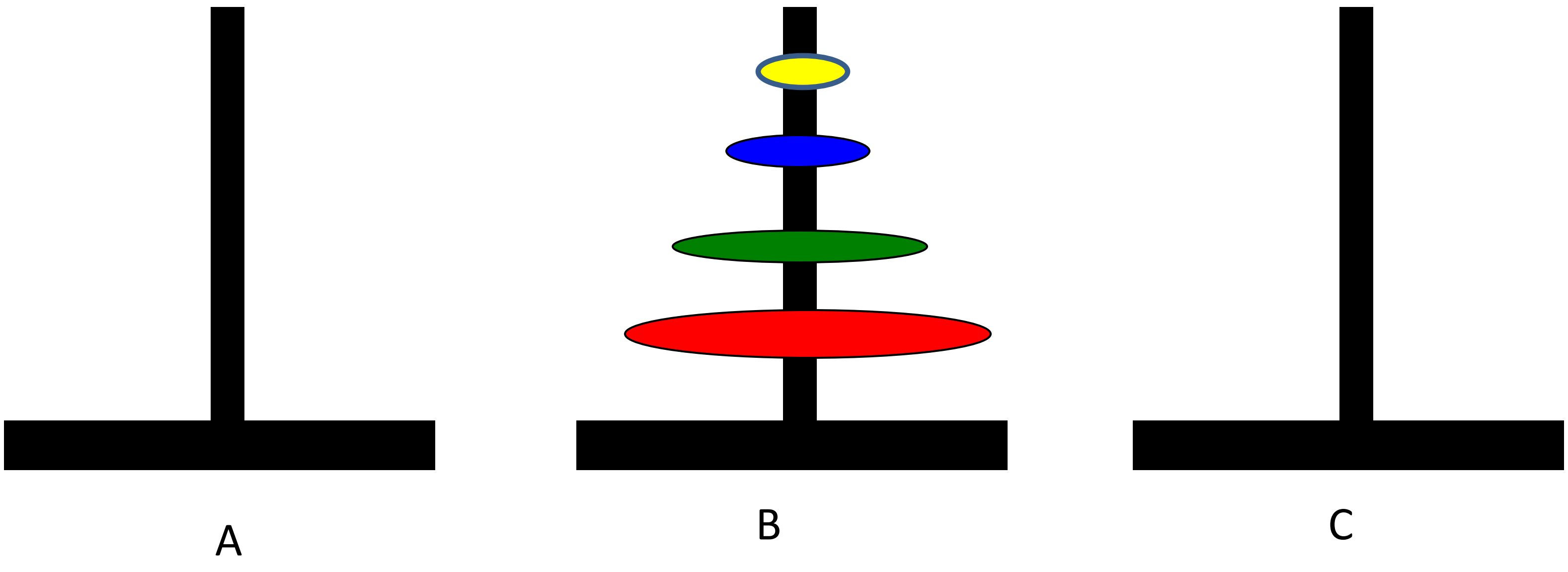
Using solution of 3 disks to solve problem with 4 disks



Using solution of 3 disks to solve problem with 4 disks



Using solution of 3 disks to solve problem with 4 disks



Recursive Algorithm

```
void Hanoi(int n, string A, string B, string C)
  {
    if (n == 1)  /* base case */
      Move(A,B);
    else { /* recursion */
      Hanoi(n-1,A,C,B);
      Move(A,B);
      Hanoi(n-1,C,B,A);
    }
  }
```

Induction

- To prove a statement $S(n)$ for positive integers n
  - Prove $S(1)$
  - Prove that if $S(n)$ is true [inductive hypothesis] then $S(n+1)$ is true.
- This implies that $S(n)$ is true for $n=1,2,3,\ldots$

Cost

- The number of moves $M(n)$ required by the algorithm to solve the n-disk problem satisfies the recurrence relation
  - $M(n) = 2M(n-1) + 1$
  - $M(1) = 1$

Guess and prove

- Calculate M(n) for small n and look for a pattern.
- Guess the result and prove your guess correct using induction.

n	M(n)
1	1
2	3
3	7
4	15
5	31

Substitution Method

- Unwind recurrence, by repeatedly replacing M(n) by the r.h.s. of the recurrence until the base case is encountered.

$M(n) = 2M(n-1) + 1$

$= 2*[2*M(n-2)+1] + 1 = 2^2 * M(n-2) + 1+2$

$= 2^2 * [2*M(n-3)+1] + 1 + 2$

$= 2^3 * M(n-3) + 1+2 + 2^2$

Geometric Series

- After k steps

  $M(n) = 2^k * M(n\text{-}k) + 1+2 + 2^2 + \ldots + 2^{n-k-1}$

- Base case encountered when k = n-1

  $M(n) = 2^{n-1} * M(1) + 1+2 + 2^2 + \ldots + 2^{n-2}$

  $= 1 + 2 + \ldots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i$

Max Sub Array Sum

- You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.
- For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

- Iterative method with two loops can solve the problem
- Cost of the solution in terms of number of addition:
- (n-1)+(n-2)+…+1=(n-1)(n-2)/2 = (n^2 – 3n + 2)/2 = O(n^2)

Solution with Divide and Conquer

- MaxSubSum(list L)
  - Divide the list into L1 and L2
  - x1=Find the MaxSubSum of L1 in recursive manner
  - x2=Find the MaxSubSum of L2 in recursive manner
  - x3=Find the MaxSubSum of list which includes last element of L1 and first element of L2
  - Return max of (x1,x2,x3)

- Base Condition: when there is single element in the list return it.

- How to get x3?
- Start from last element of left list and find max sum and in similar way start with first element of right list and find max sum then add

- $T(n)=2T(n/2)$ + cost of conquer $= 2T(n/2)+n$

  $=2(2T(n/4) +n/2) +n = 4T(n/4) + 2n$

  $=4(2T(n/8) +n/4) + 2n = 8T(n/8) +3n$

  $=2^iT(n/2^i)+i*n=n\log_2 n$

Sum of maximum of all subarrays

- **Input :** arr[] = {1, 3, 1, 7}
- **Output :** 42
- Max of all sub-arrays:
- {1} - 1
- {1, 3} - 3
- {1, 3, 1} - 3
- {1, 3, 1, 7} - 7
- {3} - 3
- {3, 1} - 3
- {3, 1, 7} - 7
- {1} - 1
- {1, 7} - 7
- {7} - 7
- Total= 1 + 3 + 3 + 7 + 3 + 3 + 7 + 1 + 7 + 7 = 42

- When array contains following **a b max c d**
- How much max will contribute?
- max X number of groups in which max is maximum element.

- Max will be maximum element of following subgroups
- max
- b max
- a b max
- max c
- b max c
- a b max c
- max c d
- b max c d
- a b max c d

- Can we come up with a formula?

- Lets say (l,r) is a range in which max is the maximum element
- Lets also say that index of max is i
- In left of max, number of elements is i-l and in right of max number of elements is r-i.
- From left side we can choose sub-array in i-l+1 way where max is included
- Like a b max
- b max
- max
- We can choose in 3 ways where max is included
- Similarly for each sub-array chosen from left side we can choose right-sub array in (r-i+1) ways
- So number of sub-array where arr[i] will contribute is (i-l+1)*(r-i+1)

```
maxSumSubarray(arr, l, r) {
        if(l==r)
                return arr[l];
        i=index_of_max(arr,l,r)
        return (arr[i]*(r-i+1)*(i-l+1) +
        maxSumSubarray(arr, l, i-1) +
        maxSumSubarray(arr, i+1, r))
}
```

HomeWork

- Develop a recursive code for trinary search. A search would be called trinary when input list is divided in three sub-lists of size n/3, (n-n/3)/2, n-(n-n/3)/2 -n/3 and search is performed in appropriate sub-list.
- Make a comparison of worst case cost for binary search and trinary search for n=10-100 and find if you can conclude which one is better.
- Input : A list of sorted integer in increasing order and an element to be searched
- Output: position of the element in the list if it is found otherwise -1.