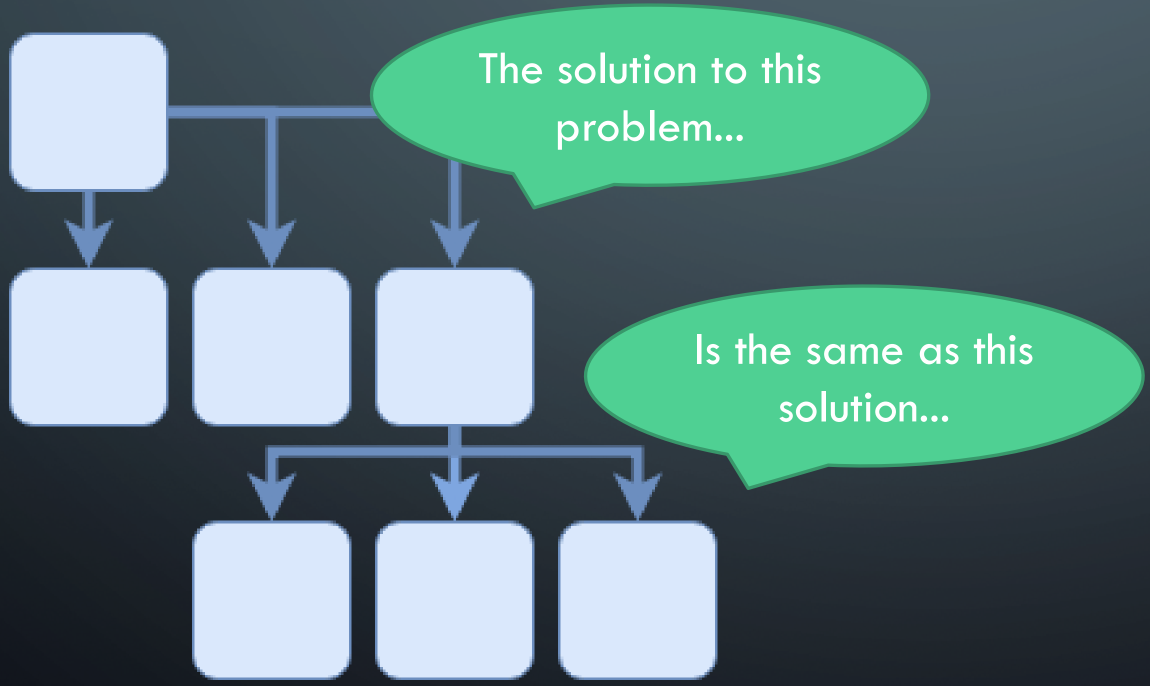
LECTURE 3

RECURSION

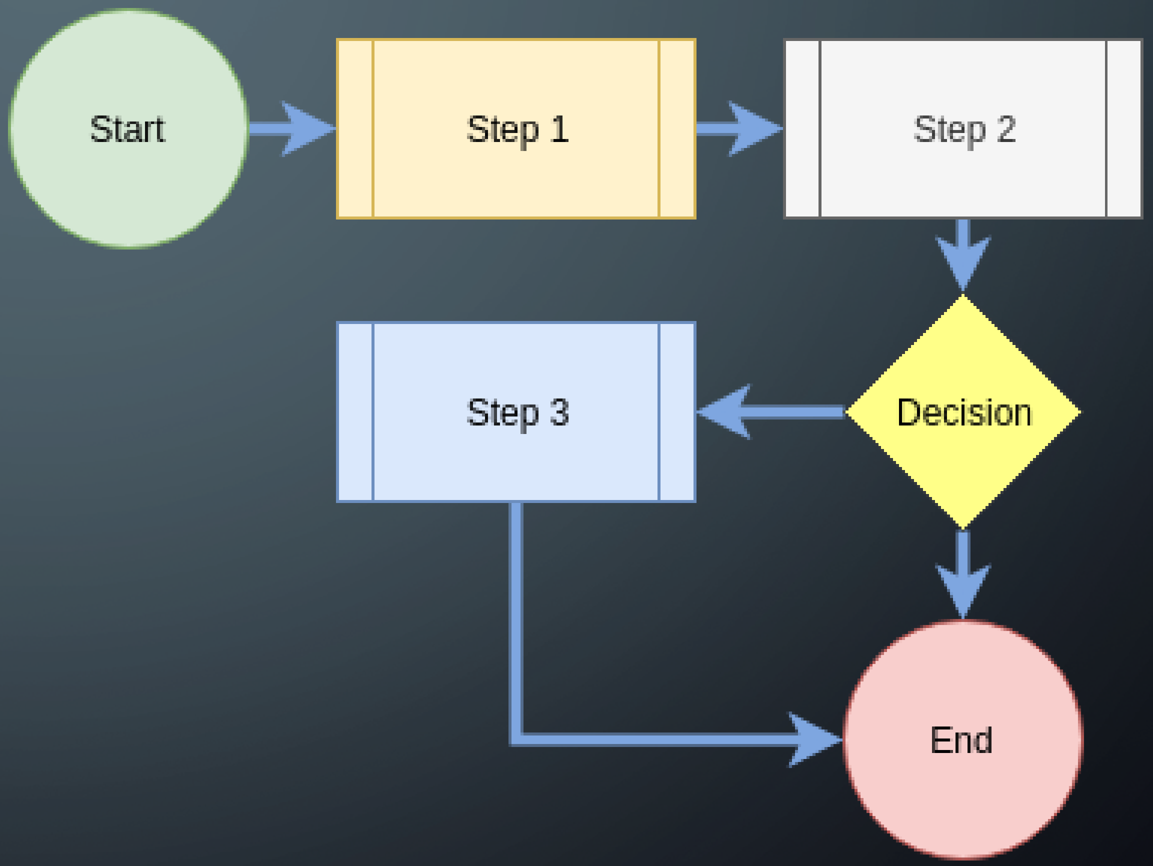
Recursion is a powerful and elegant way to apply a common solution to solve a larger problem.

- If we have a problem that can be divided into similar subtasks, there is a potential to apply recursion
- The goal is to write one solution which can be applied across all tasks



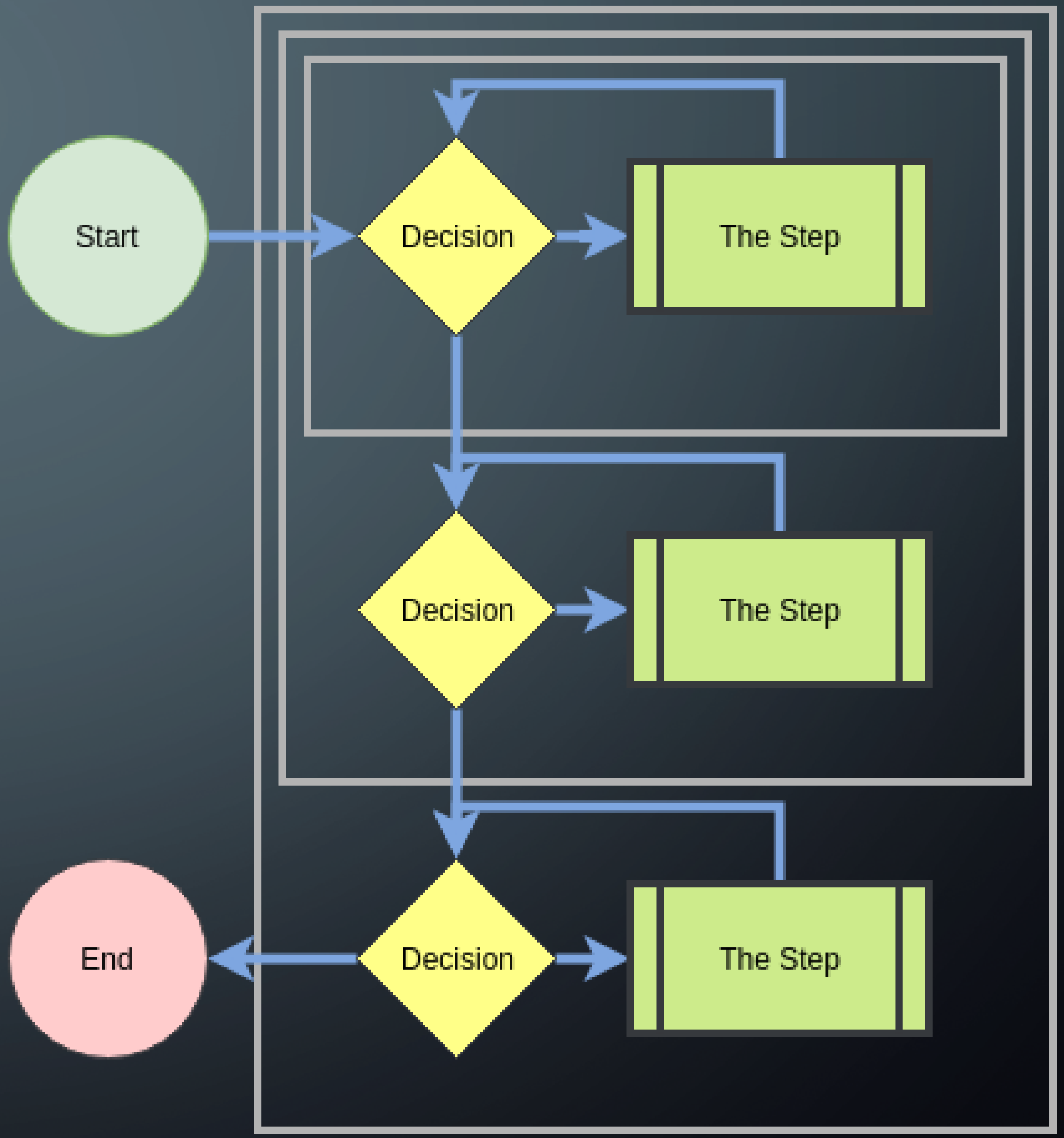
Comparison: Iterative, Acyclic Algorithm

- Iterative flow of steps
  - No cycles (looping back)
  - Set amount of operations
- Steps are distinct
  - The task of Step 1 cannot be solved by Step 3
- Not conducive to division of work
  - Large inputs must process each item through the algorithm



Comparison: Recursive Algorithm

- Nested flow
  - The same task is reused for each box/stage
- Big workloads can be broken up
  - Each division makes the workload per nesting smaller
  - Once all divisions are small enough, the same task is reused
  - Inner boxes feed their results back upstream



Example: The factorial operation n!

- Simple enough to follow…
- Note: we have the same basic process for each iteration
  - Check if we are done
  - Decrement
  - Multiply
- How can we make this recursive?

```
int factorial(int n) {
  int res = n;
  while (n > 1) {
    n = n - 1;
    res = res * n;
  }
  return res;
}
```

Example: The factorial operation n! with recursion

- Consider the definition of the factorial:
  - n! = n * (n-1) * (n-2) * … * 3 * 2 * 1
  - n! = n * [(n-1) * (n-2) * … * 3 * 2 * 1]
  - n! = n * (n-1)!
- We can define the factorial in terms of itself!
  - The factorial of n is n times the factorial of (n-1)...
  - The factorial of (n-1) is (n-1) times the factorial of (n-2)…
- This relationship is called a recurrence-relation, and we can use it to directly form a recursive solution

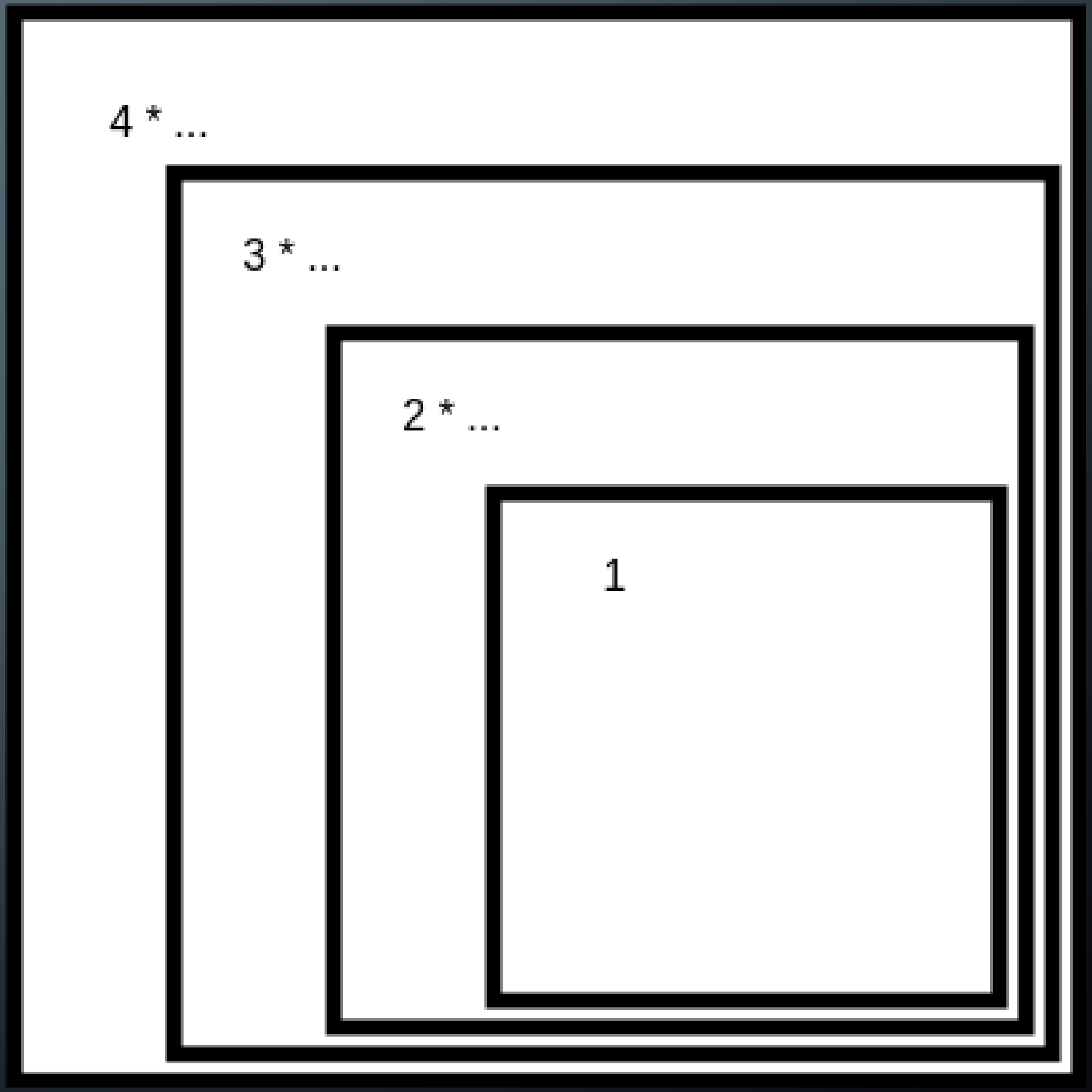
Example: The factorial operation n! with recursion

- Now, we use our recurrence relation directly
    - Each step passes a smaller task to the nested call downstream
    - Each step commits to solving the nth multiplication to pass upstream

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

Example: The factorial operation n! with recursion

- Graphical perspective…
    - Each box takes as input the output of the box inside it
    - Each box takes its number, multiplies it by its input, and passes the output upstream



Important Parts of Recursive Algorithms

- All recursive procedures will at some point invoke themselves after some modification of their inputs
  - Our factorial procedure called itself to solve the problem!
- It is important that each step gets you closer to the solution somehow. Every step should reduce the total workload somehow
  - Each step of the factorial took care of multiplying one number
- There must be some way to tell when you are done…
  - The base case of our factorial program was n=1
  - Your program will happily run forever if you forget this…until you get a stack overflow…

What's Missing?

- What three things define a recursive procedure?
  - Are we invoking the procedure from itself?
  - Do we have a base case to check?
  - Are we advancing toward a solution?

```
int sum(int n){
  return n*(n-1)/2;
}
```

What's Missing?

- What three things define a recursive procedure?
  - Are we invoking the procedure from itself?
  - Do we have a base case to check?
  - Are we advancing toward a solution?

```
int gcd(int n, int m){
   if(n==0){
      return m;
   }
   else{
      return gcd(n,m);
   }
}
```

EUCLIDS GCD ALGORITHM

Algorithm *Euclid.* Given two positive integers m and n, find the greatest common divisor, that is, the largest positive integer that evenly divides both m and n

A.0 If m < n, swap m and n

A.1 Divide m by n and let r be the remainder.

A.2 If r = 0, terminate; n is the answer.

A.3 Set m to n, n to r, and go back to step A.1

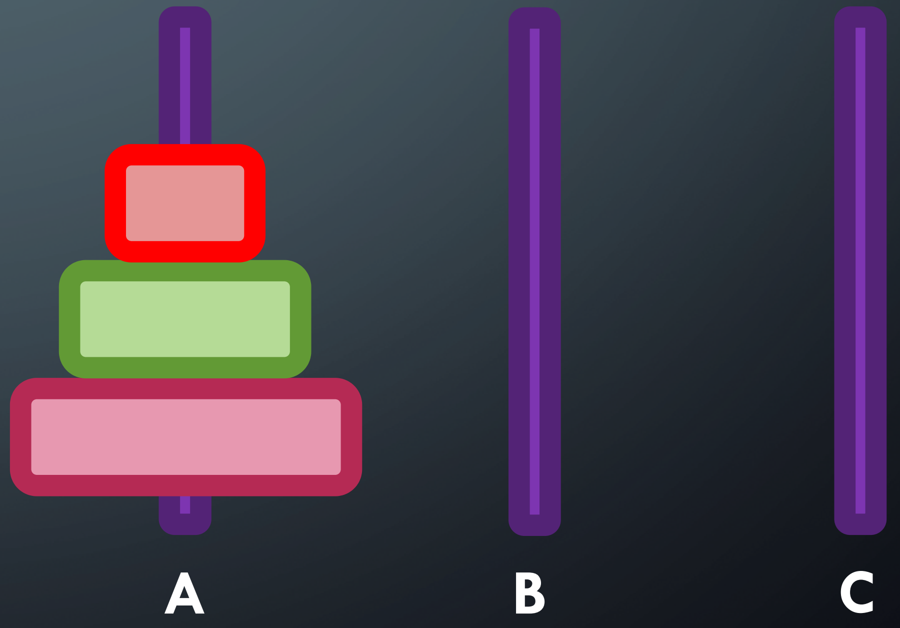
```
int gcd(int n, int m){
   if (m < n)
      swap (m,n);
   if(n==0)
      return m;
   if (m%n == 0)
      return n;
   int r = m/n;
   return gcd(n,r);
}
```

Putting recursion to use on more complicated problems

- We have seen how recursion applied for simple tasks, data structures, and array processing
  - So far, the simple examples haven't provided a compelling case for recursion over the iterative approach
- The last example is deceptively powerful: what if our search space is HUGE?
  - Consider the task of searching a database to verify someone's credit card number is valid
  - Each credit card company must search about 1 Billion numbers over a phone line, while you're waiting in line at Einstein's...
  - Dividing the search space in half each time is an enormous advantage!

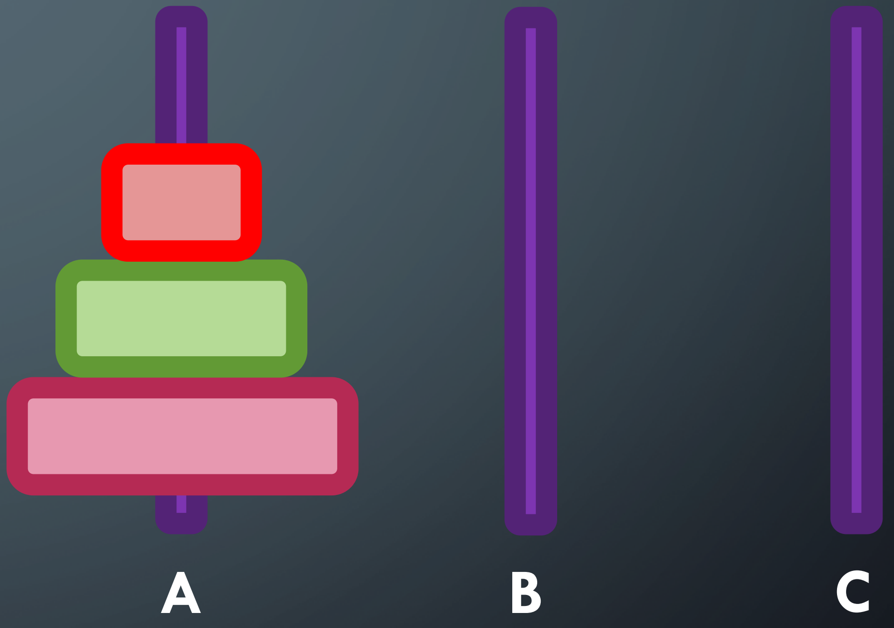
Ancient solutions for ancient problems

- Recursion often offers an elegant solution for difficult problems
- The classic "Towers of Hanoi" problem is one example
- Goal:
  - Move N discs from the source peg A to the target peg C
  - At all times, the discs must be ordered from smallest to largest, top to bottom



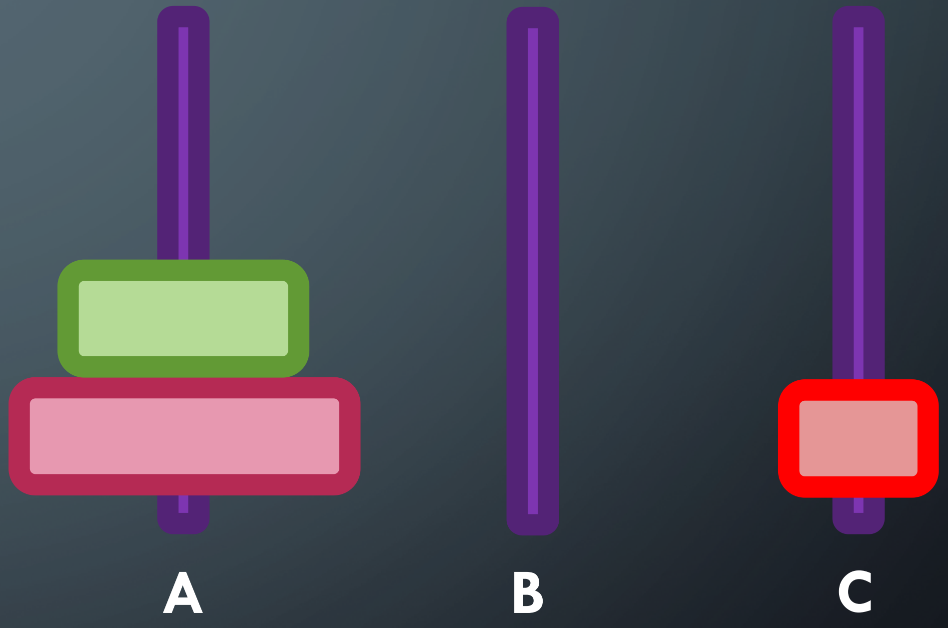
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:



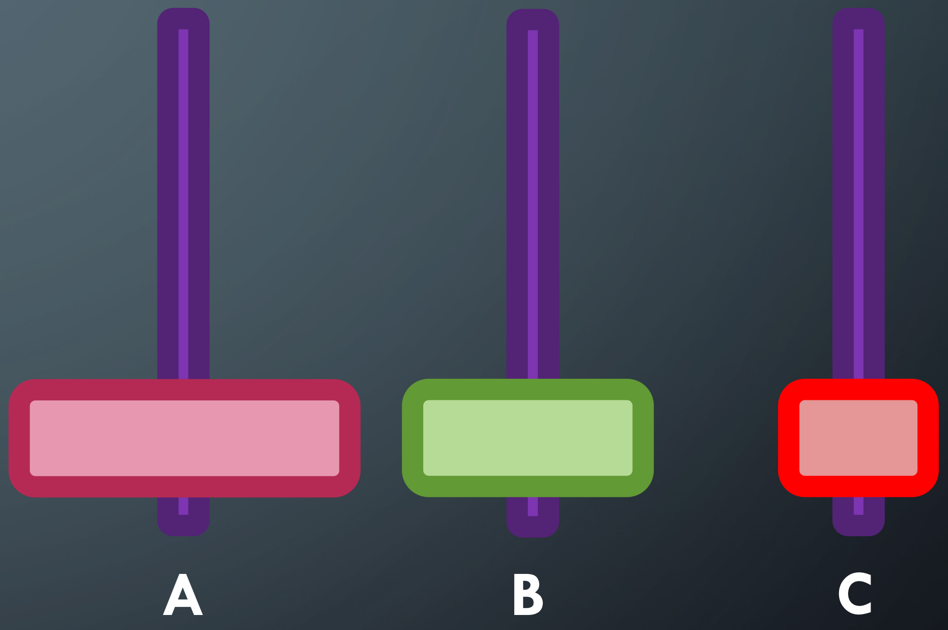
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:
    - Move red to target peg C



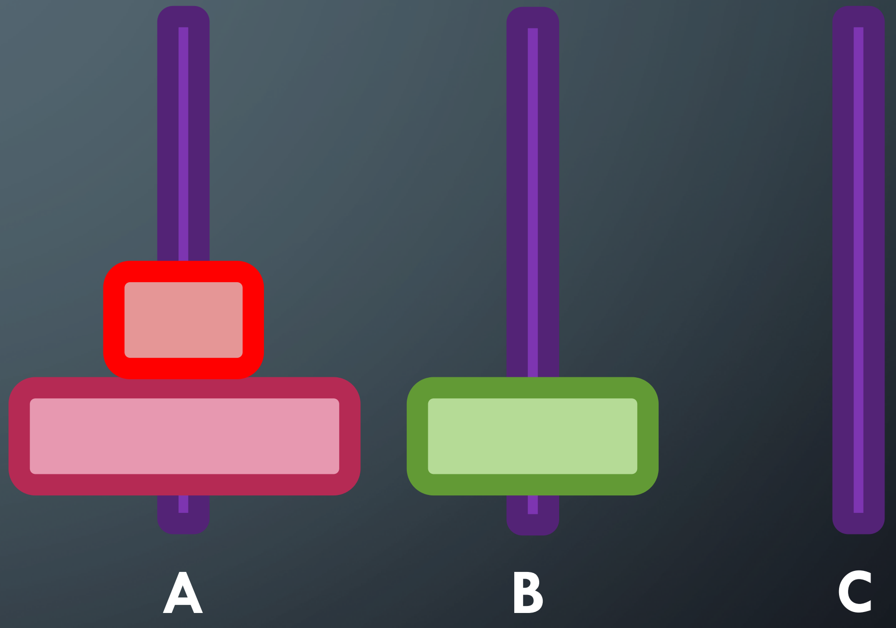
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:
    - Move red to target peg C
    - Move green to spare peg B



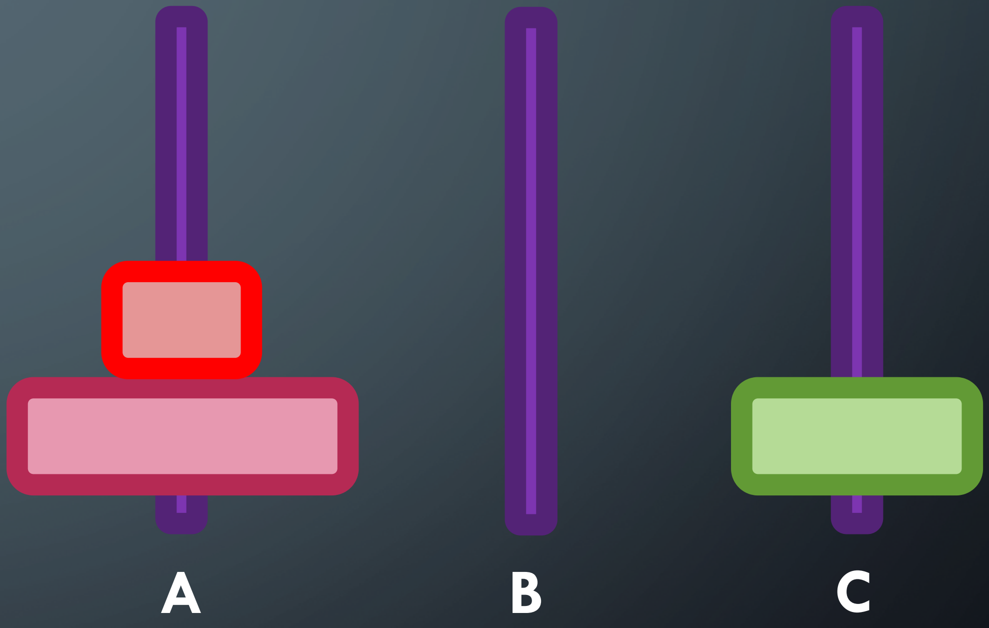
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:
    - Move red to target peg C
    - Move green to spare peg B
    - Move red to source peg A



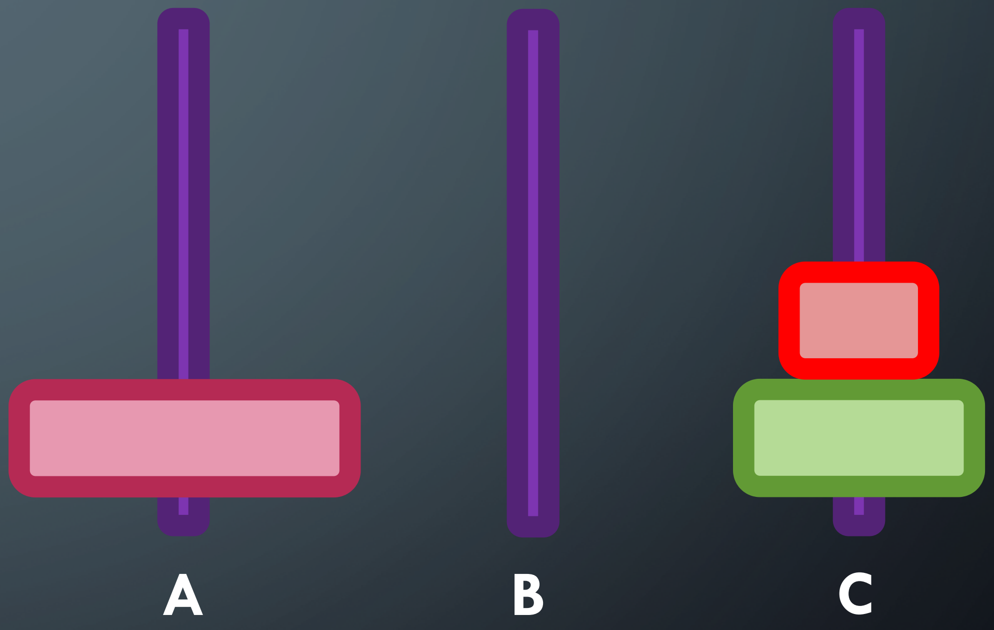
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:
    - Move red to target peg C
    - Move green to spare peg B
    - Move red to source peg A
    - Move green to target peg C



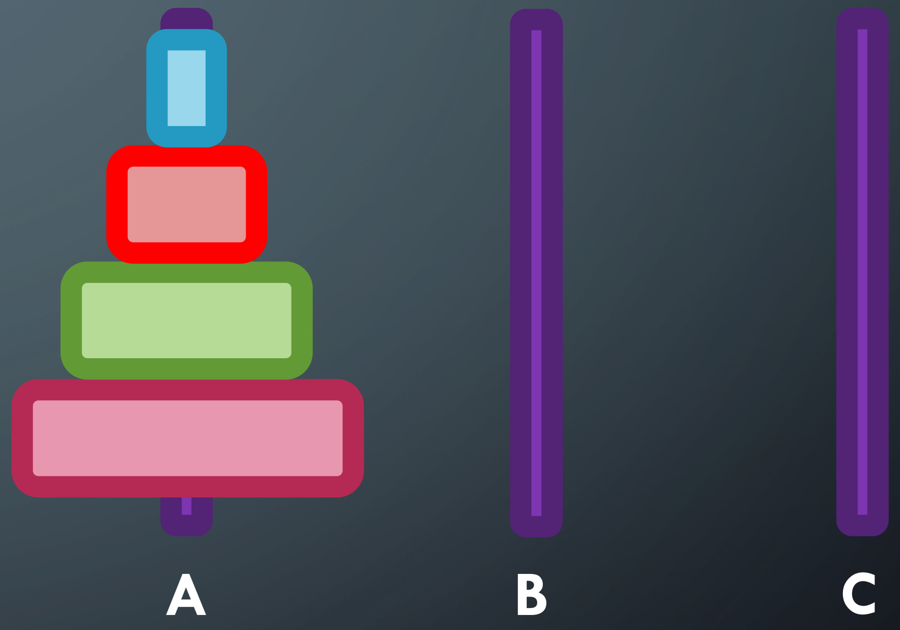
Ancient solutions for ancient problems

- A few observations:
  - A single disc can be moved to any peg with no discs or larger discs
  - To move a pair of stacked disks, we can follow this procedure:
    - Move red to target peg C
    - Move green to spare peg B
    - Move red to source peg A
    - Move green to target peg C
    - Move red to target peg C



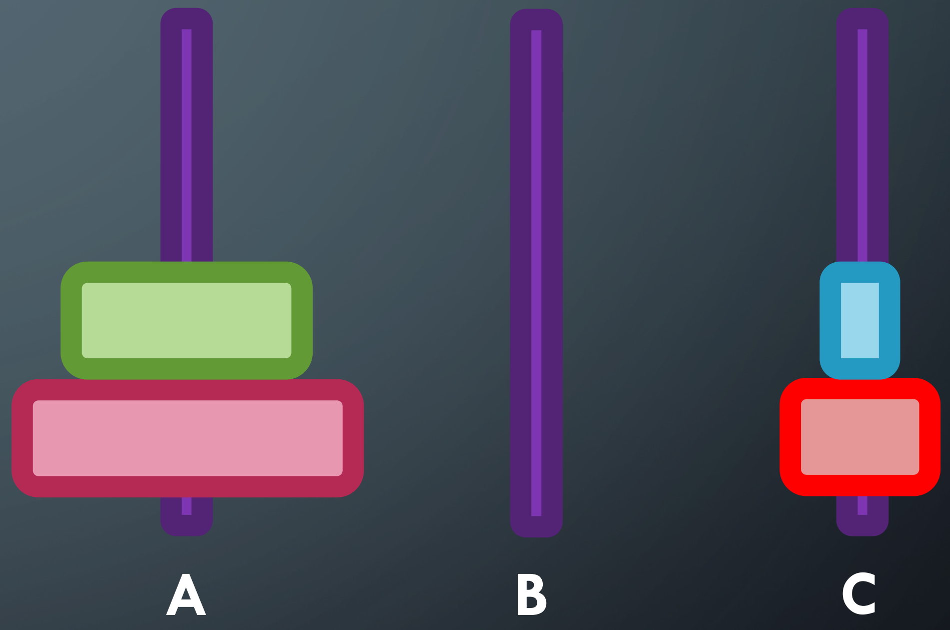
Ancient solutions for ancient problems

- How can we use recursion to help us here?



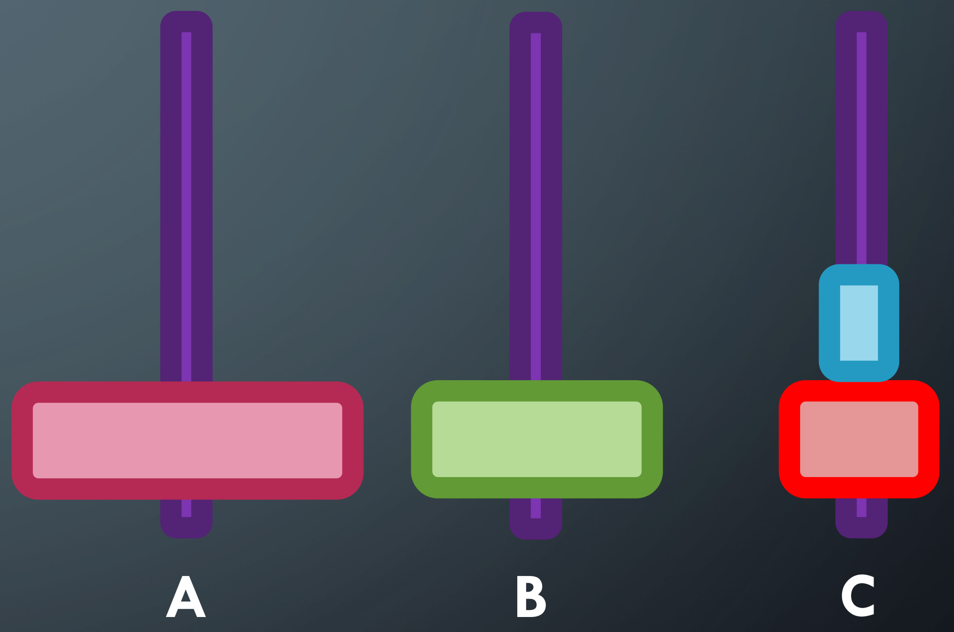
Ancient solutions for ancient problems

- How can we use recursion to help us here?
- Another observation: if we have moved a pair to peg C



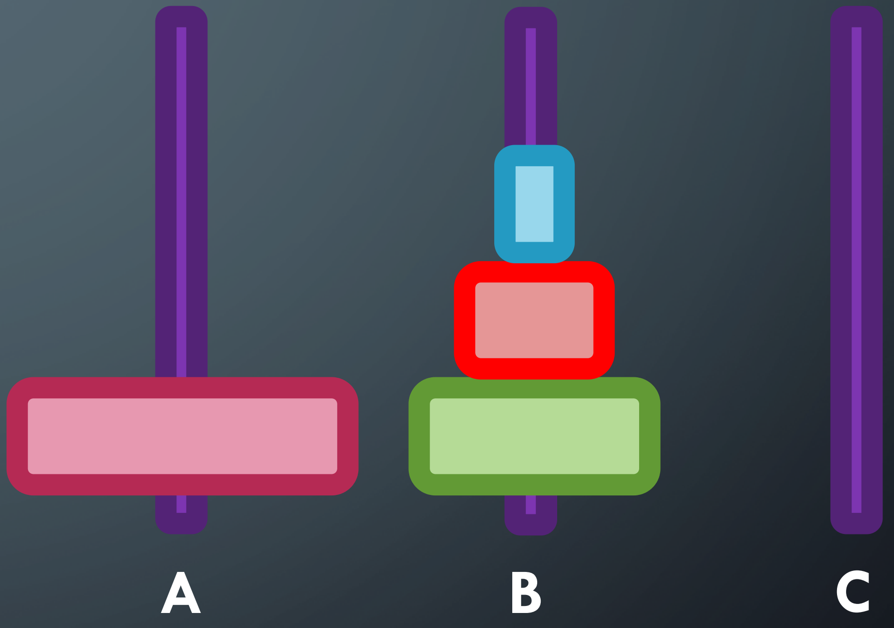
Ancient solutions for ancient problems

- How can we use recursion to help us here?
- Another observation: if we have moved a pair to peg C, we can move a single disc (green) to peg B



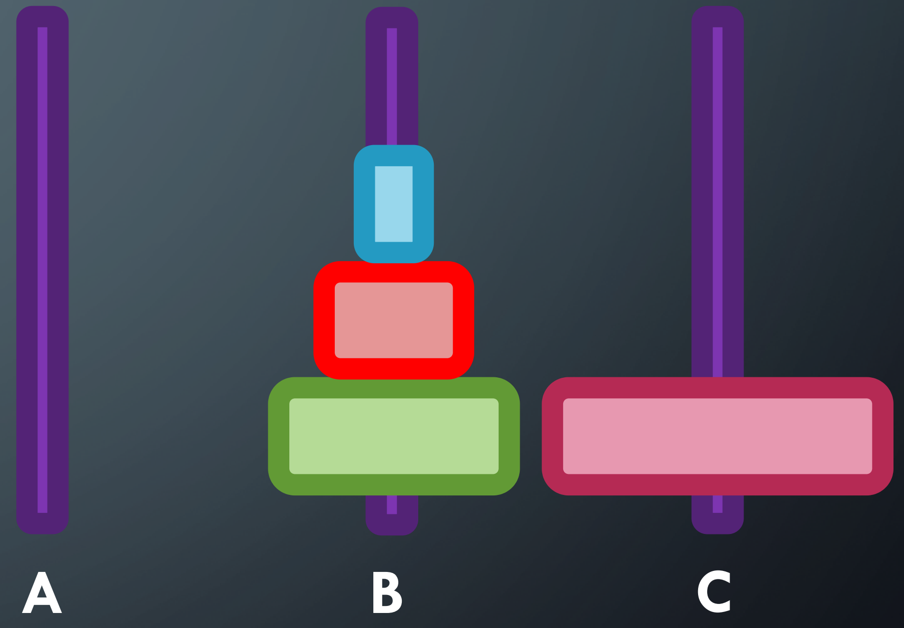
Ancient solutions for ancient problems

- How can we use recursion to help us here?
- Another observation: if we have moved a pair to peg C, we can move a single disc (green) to peg B
  - We can then move the pair back on top of green in peg B



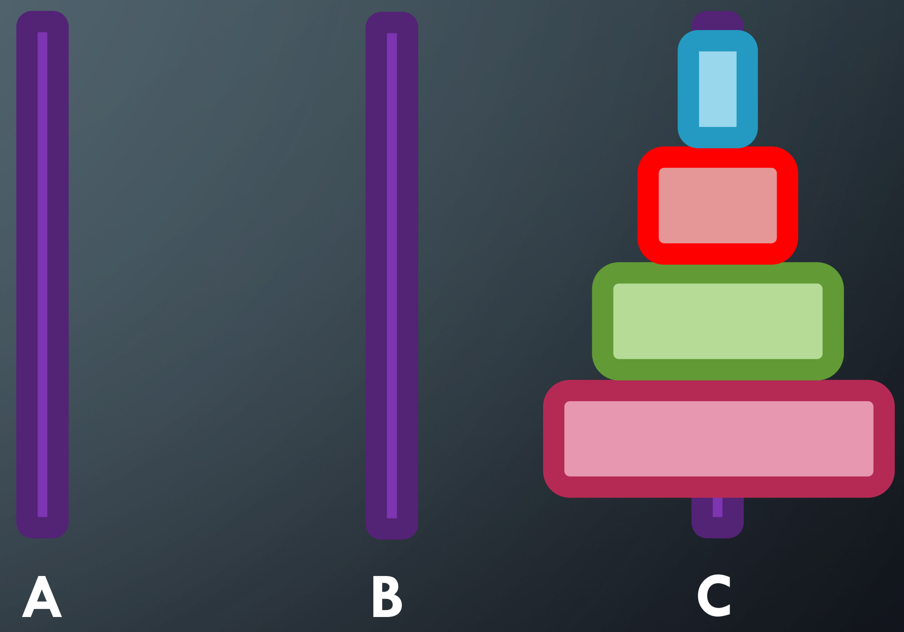
Ancient solutions for ancient problems

- How can we use recursion to help us here?
- Another observation: if we have moved a pair to peg C, we can move a single disc (green) to peg B
  - We can then move the pair back on top of green in peg B
  - We can then move the pink disc to peg C



Ancient solutions for ancient problems

- How can we use recursion to help us here?
- Another observation: if we have moved a pair to peg C, we can move a single disc (green) to peg B
  - We can then move the pair back on top of green in peg B
  - We can then move the purple disc to peg A
  - We can repeat the entire process...



Recursive Solution to Towers of Hanoi

- This approach will print the steps required to solve for N discs
- The amount of moves required is given by 2 to the (N-1) power

```
void Towers(char A, char B, char C, int n) {
  if(n==1) {
    cout << "Move disc 1 from "
         << A  << " to " << B << endl;
  }
  else {
    Towers(A,C,B,n-1);
    cout << "Move disk " << n
         << " from peg " << A
         << " to " << B << end;
    Towers(B,A,C,n-1);
  }
}
```

Do we always want to use recursion?

- There are limitations here – every recursive call requires memory for the function call and its variables on the stack…
- For very deep recursion, this will cause a stack overflow. For less powerful hardware, this happens much sooner…
- The language matters – Haskell and Lisp are designed to use recursion for nearly everything! They use lazy evaluation to keep things fast. Some languages handle recursion more efficiently than others…
- So we have a tradeoff here: if recursion offers a sufficiently better solution, we should try to use it if we have the memory to do so… can the compiler help?