MORE RECURSION: PERMUTATIONS AND TOWERS OF HANOI

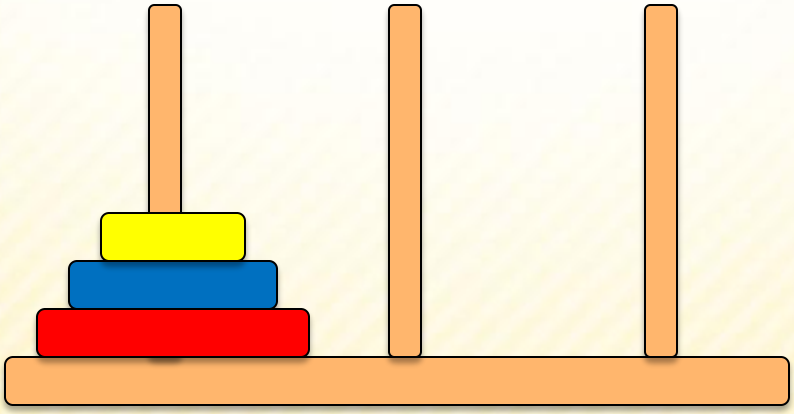
COP 3502

TOWERS OF HANOI

COP 3502

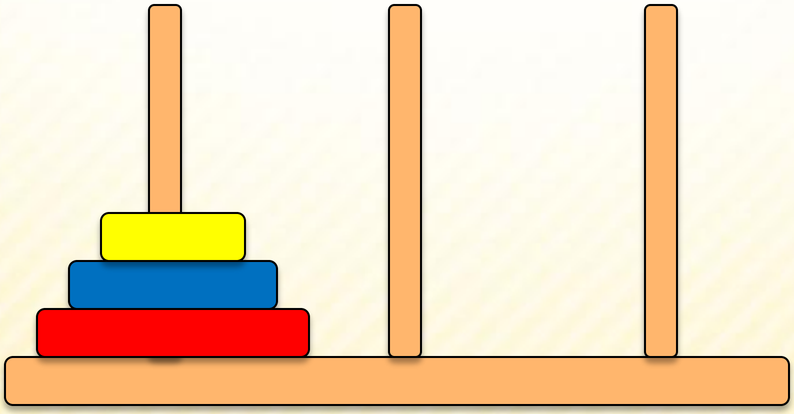
Recursion – Towers of Hanoi

- The Towers of Hanoi
  - Is a mathematical puzzle that has a classic recursive solution that we are going to examine.
  - The puzzle was invented by the French mathematician Edouard Lucas, based upon a legend:
    - In an Indian temple there contains three posts surrounded by 64 golden disks.
    - The monks have been moving the disks according to the puzzle rules since the beginning of time.
    - And according to the legend, when the last move of the puzzle is completed, the world will end.



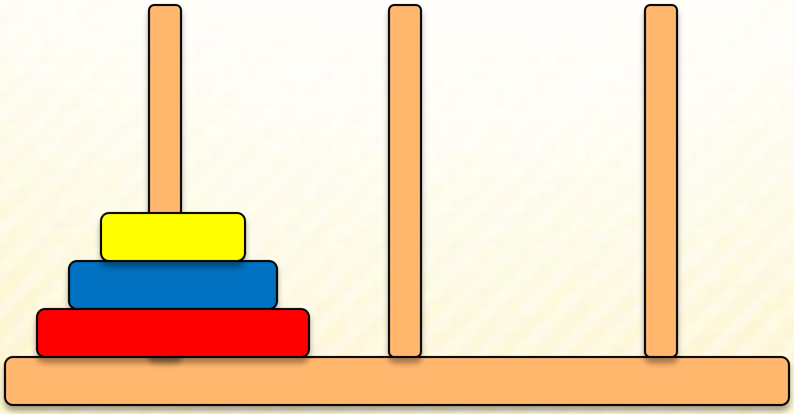
Recursion – Towers of Hanoi

- The Towers of Hanoi
  - The goal is to move all disks from Tower#1 to Tower#3.
  - The rules are:
    - You can only move ONE disk at a time
    - And you can NEVER put a bigger disk on top of a smaller disk.



Recursion – Towers of Hanoi

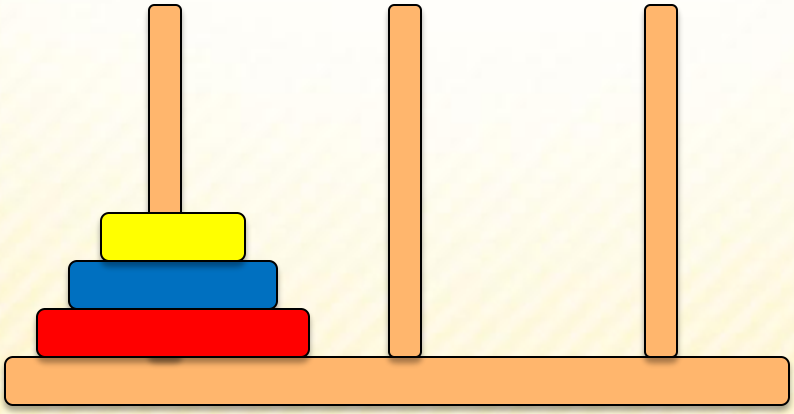
- The Towers of Hanoi
  - Coming up with a Recursive Solution:
    - Clearly an tower with more than 1 disk must be moved in pieces.
    - We know that the bottom disk needs to moved to the destination tower.
      - In order to do that we need to move all disks above the bottom disk to the intermediate tower.
      - This leads to our recursive solution!



Recursion – Towers of Hanoi

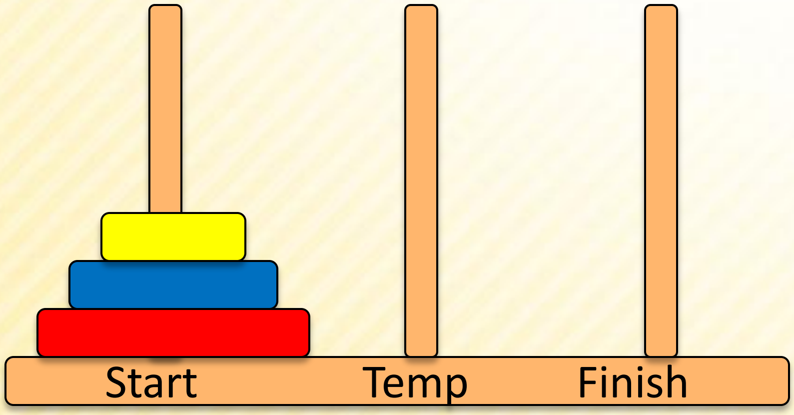
- The Towers of Hanoi
  - Solution:
    - Regardless of the number of disks, we know we have to do the following steps:
      - The bottom disk needs to be moved to the destination tower

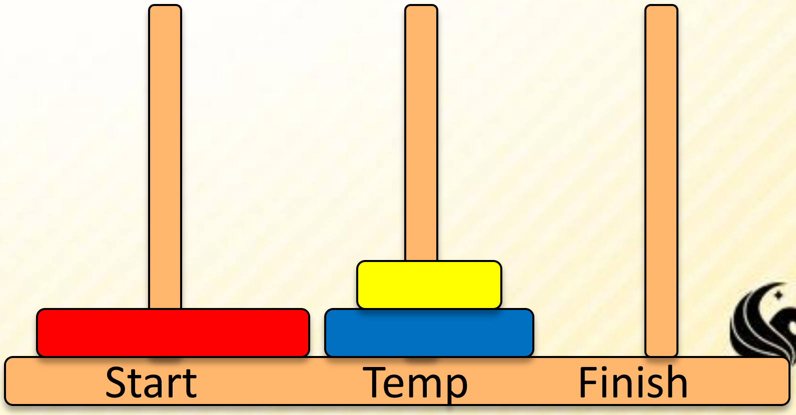
      1) So step 1 must be to move all disks above the bottom disk to the intermediate tower.
      2) In step 2, the bottom disk can now be moved to the destination tower.
      3) In step 3, the disks that were initially above the bottom disk must now be put back on top of the destination tower.



Recursion – Towers of Hanoi

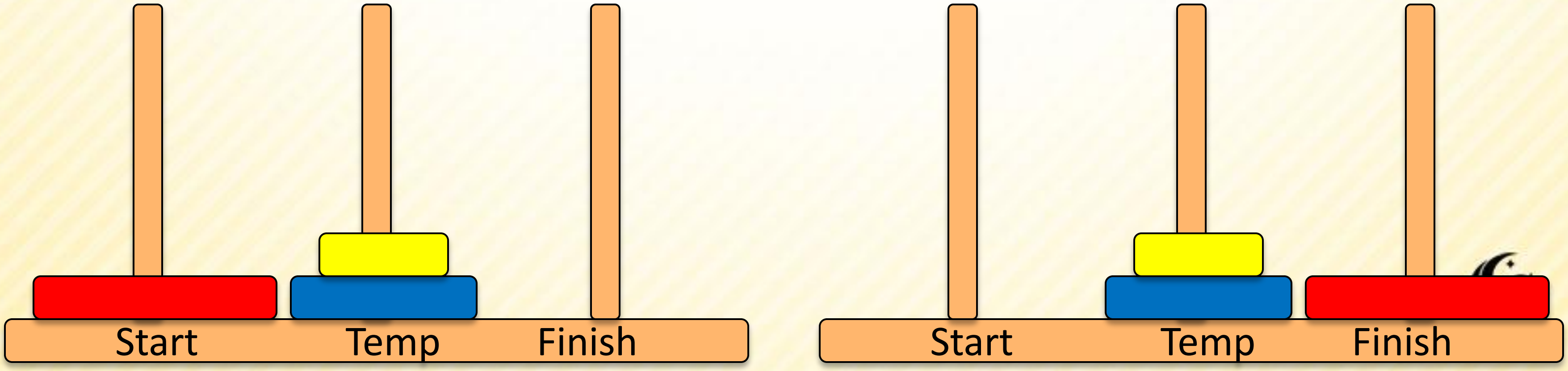
- The Towers of Hanoi
  - Let’s look at the problem with only 3 disks.
  - Solution:
    - Step 1:
      - Move top 2 disks to temp
        - » we would have to solve this recursively, since we can only move 2 disks at a time.
        - » We’re going to assume that we know how to do the 2 disk problem (since this is solved recursively), and continue to the next step.





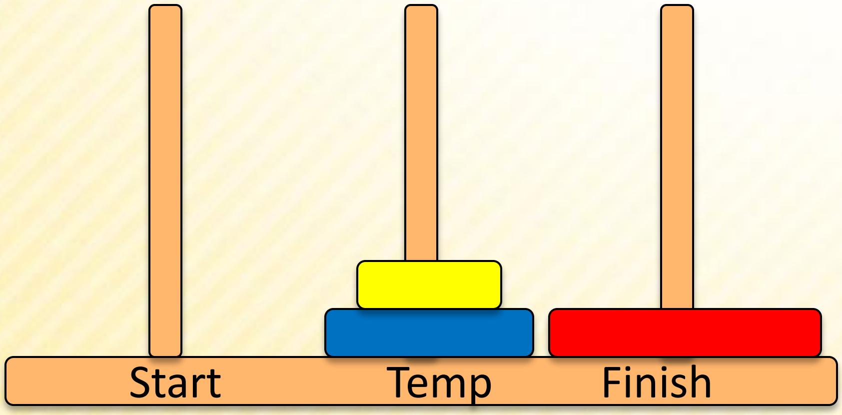
Recursion – Towers of Hanoi

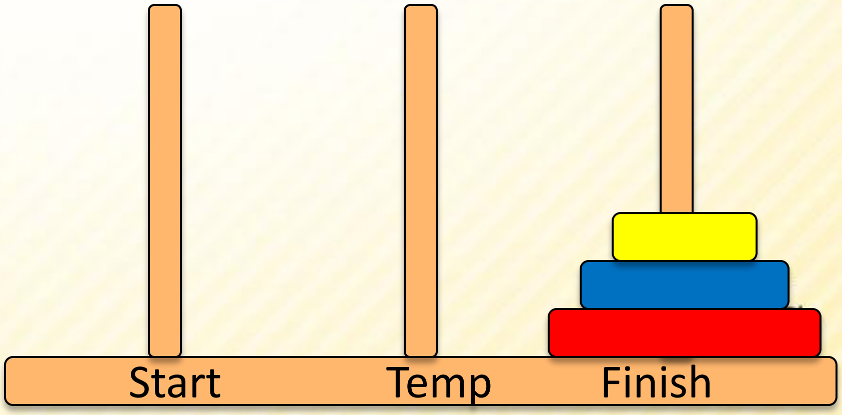
- The Towers of Hanoi
  - Let’s look at the problem with only 3 disks.
  - Solution:
    - Step 2:
      - Move the last single disk from start to finish
      - Moving a single disk does not use recursion, and does not use the temp tower.
      - (In our program, a single disk move is represented with a print statement.)

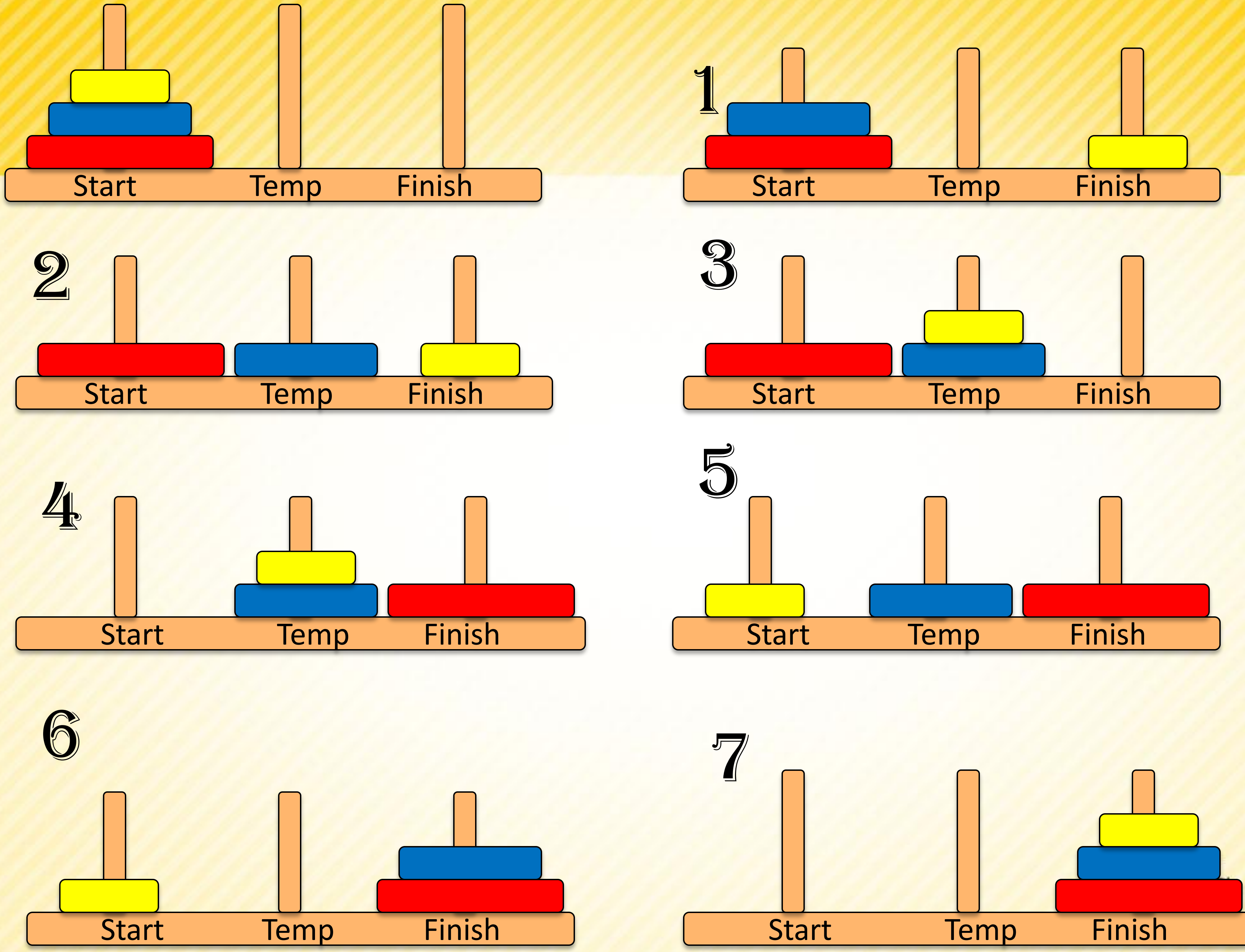


Recursion – Towers of Hanoi

- The Towers of Hanoi
  - Let's look at the problem with only 3 disks.
  - Solution:
    - Step 3:
      - Last step – Move the 2 disks from Temp to Finish
        - This would be done recursively.

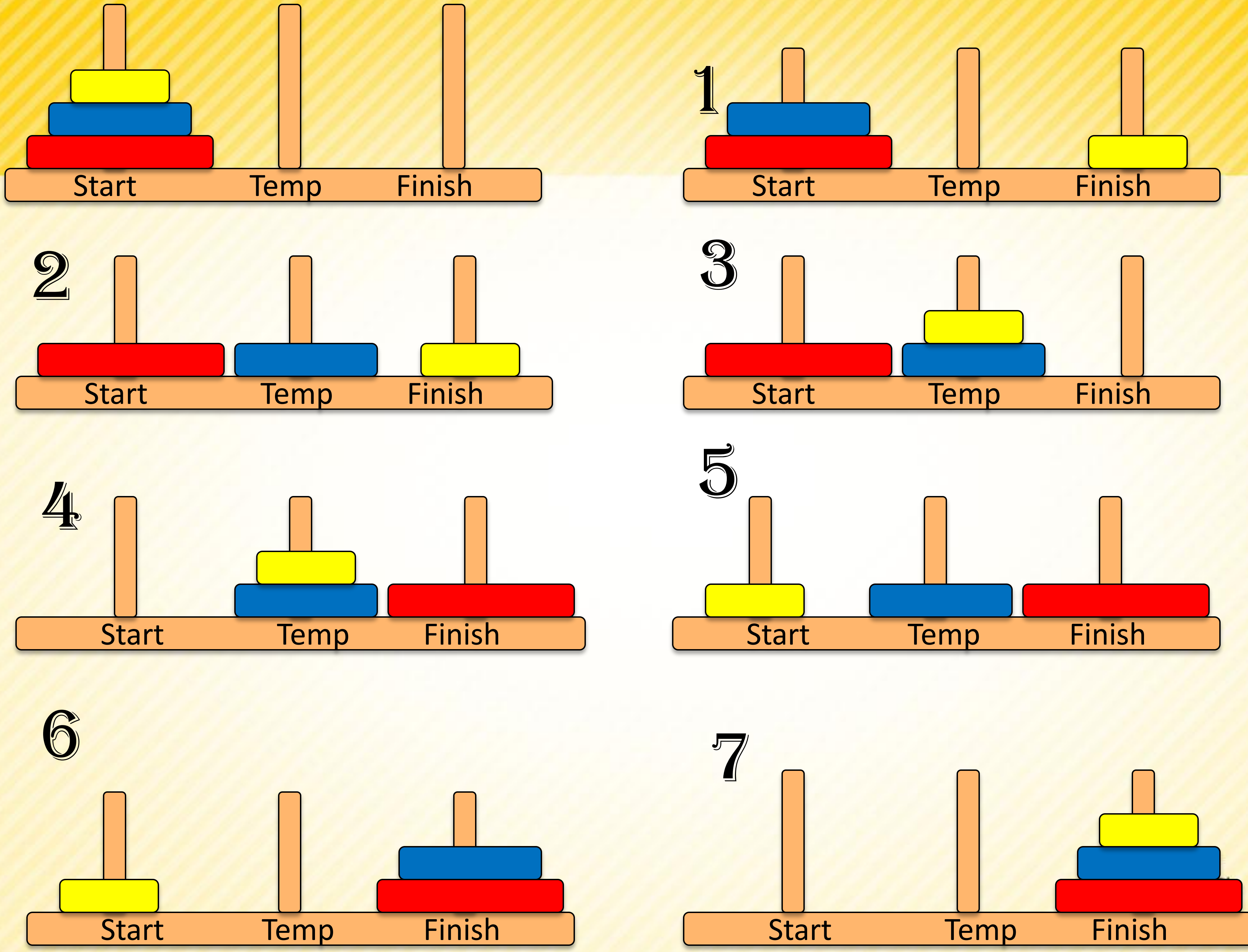


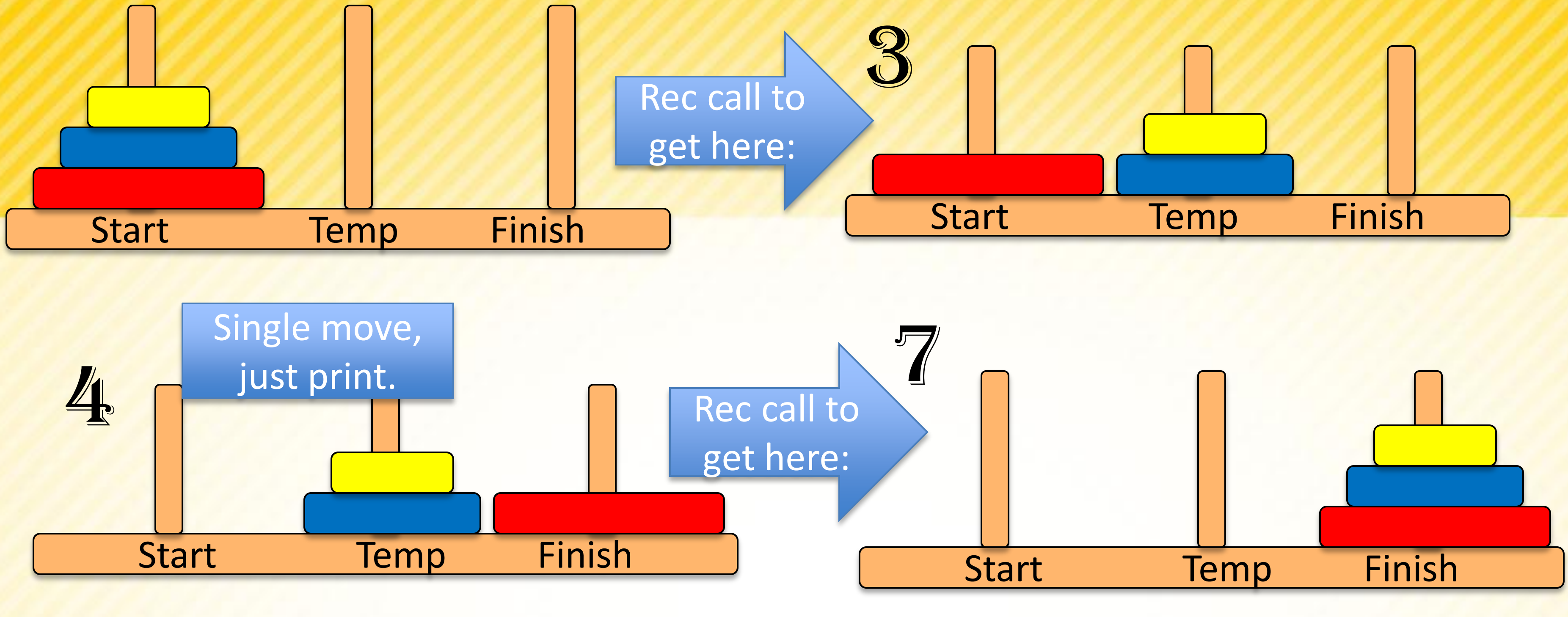



Start
Temp
Finish
1
Start
Temp
Finish
2
Start
Temp
Finish
3
Start
Temp
Finish
4
Start
Temp
Finish
5
Start
Temp
Finish
6
Start
Temp
Finish
7
Start
Temp
Finish

Recursion – Towers of Hanoi

- Number of Steps:
  - 3 disks required 7 steps
  - 4 disks would requre 15 steps
  - We get n disks would require $\mathbf{2^n - 1}$ steps
    - HUGE number


Start
Temp
Finish
1
Start
Temp
Finish
2
Start
Temp
Finish
3
Start
Temp
Finish
4
Start
Temp
Finish
5
Start
Temp
Finish
6
Start
Temp
Finish
7
Start
Temp
Finish




```
void doHanoi(int n, char start, char finish, char temp) {
        if (n==1) {
                printf("Move Disk from %c to %c\n", start,
finish);
        }
        else {
                doHanoi(n-1, start, temp, finish);
                printf("Move Disk from %c to %c\n, start finish);
                doHanoi(n-1, temp, finish, start);
        }
}
```

```
// Function Prototype
void doHanoi(int n, char start, char finish, char temp);

void main() {
        int disk;
        int moves;
        printf("Enter the # of disks you want to play with:");
        scanf(%d", &disk);
        // Print out the # of moves required
        moves = pow(2, disk)-1;
        printf("\nThe # of moves required is = %d \n", moves);
        // Show the moves using doHanoi
        doHanoi(disk, 'A', 'C', 'B');
}
```

Permutations

- The permutation problem is as follows:
  - Given a list of items, list all the possible orderings of those items.
  - For example, here are all the permutations of CAT:
    - CAT
    - CTA
    - ACT
    - ATC
    - TAC
    - TCA

Permutations

- There are several different permutation algorithms,
  - but since we're focusing on recursion in this course, a recursive algorithm will be presented.
  - (Feel free to come up with or research an iterative algorithm on your own)

Recursive Permutation Algorithm

- The idea is as follows:
  - In order to list all the permutations of CAT, we can split our work into three groups of permutations:
    1) Permutations that start with C.
    2) Permutations that start with A.
    3) Permutations that start with T.

  - The recursion comes in here:
    - When we list all permutations that start with C, they are nothing but strings formed by attaching C to the front of ALL permutations of “AT”.
    - This is nothing but another permutation problem!!!

Recursive Permutation Algorith

- Number of recursive calls
  - Often when recursion is taught, a rule of thumb is:
    - “recursive functions don’t have loops”
  - Unfortunately, this rule of thumb is not always true!
    - An exception to this rule is the permutation algorithm.

Recursive Permutation Algorith

- Number of recursive calls
  - The problem is the number of recursive calls is variable.
  - In the CAT example
    - 3 recursive calls were needed
  - BUT, what if we were permuting the letters in the word, "COMPUTER"?
    - Then 8 recursive calls (1 for each possible starting letter) would be needed.

Recursive Permutation Algorith

- Number of recursive calls
  - In other words…
  - We need a loop in the algorithm
    - for (each possible starting letter)
      - list all permutations that start with that letter
  - What is the terminating condition?
    - Permuting either 0 or 1 element.
      - In these cases there's nothing to permute
    - In our code, we will use 0 as our terminating condition.

Recursive Permutation Algorithm

- The Permutation algorithm:
  - As we have seen in previous examples
    - some recursive functions take in an extra parameter compared to their iterative implementation
    - This is usually used to keep track of the number of iterations left until the base case.
  - This is the case for our permutation algorithm
    - Shown in the following function…

Recursive Permutation Algorithm

```
// Pre-condition:   str is a valid C String, and k
//                  is non-negative and <= the
//                  length of str.
// Post-conditions: All of the permutations of str with
//                  the first k characters fixed in
//                  their original positions are
//                  printed.  Namely, if n is the lenth
//                  of str, then (n-k)! permutations are
//                  printed.
void RecursivePermute(char str[], int k);
```

- So k refers to the **<u>first k characters that are fixed in their original positions.</u>**

Recursive Permutation Algorithm

```
// Pre-condition:    str is a valid C String, and k
//                   is non-negative and <= the
//                   length of str.
// Post-conditions: All of the permutations of str with
//                   the first k characters fixed in
//                   their original positions are
//                   printed.  Namely, if n is the lenth
//                   of str, then (n-k)! permutations are
//                   printed.
void RecursivePermute(char str[], int k);
```

- So we terminate when k is equal to the length of the string, str
  - **<u>This means:</u>**
    - **<u>If k is equal to the length of the actual string, and all k values are fixed, there's nothing left to permute</u>**
    - So we just print out that permutation

Recursive Permutation Algorithm

```
// Pre-condition:   str is a valid C String, and k
//                  is non-negative and <= the
//                  length of str.
// Post-conditions: All of the permutations of str with
//                  the first k characters fixed in
//                  their original positions are
//                  printed.  Namely, if n is the lenth
//                  of str, then (n-k)! permutations are
//                  printed.
void RecursivePermute(char str[], int k);
```

- If we do NOT terminate:
  - We want a loop that tries each character at index k.

Recursive Permutation Algorithm

- The recursive algorithm:

```
void RecursivePermute(char str[], int k) {
       int j;

       // Base-case:  All fixed, so Print!
       if (k == strlen(Str))
              pringf("%s\n", str);
       else {
              // Try each letter in spot j
              for (j=k; j<strlen(Str); j++) {
                     // Place next letter in spot k.
                     ExchangeCharacters(str, k, j);

                     // Pring all with spot k fixed.
                     RecursivePermute(str, k+1);

                     // Put the old char back.
                     ExchangeCharacters(str, j, k);
              }
       }
}
```

Recursive Permutation Algorithm

- The main loop within the recursive algorithm:

```
for (j=k; j<strlen(Str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

- How do we get the different characters in the first position?
  - (The ‘C’, ‘A’, ‘T’ , in our CAT example)

Recursive Permutation Algorithm

- The main loop within the recursive algorithm:

```
for (j=k; j<strlen(Str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

- The ExchangeCharacters function:
  - Takes in str, and swaps 2 characters within that string (at index k and index j)

Recursive Permutation Algorithm

- This function will swap the characters for us,
  - Letting each character have a turn at being the 1st character in the sub-string

```
// Pre-condition: str is a valid C String and i,j are
                  valid indices to that string.
// Post-condition:  The characters at i and j, will
                    be swapped in str.
void ExchangeCharacters(char str[], int i, int j) {
       char temp = str[i];
       str[i] = str[j];
       str[j] = temp;
}
```

Recursive Permutation Algorithm

- The main loop within the recursive algorithm:

```
for (j=k; j<strlen(Str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

- So after we swap positions, we swap back so we can continue looping through the rest of the possible characters at position k.

Recursive Permutation Algorithm

- Recursive Permutation code in detail:
  - 2 parameters to the function

1) The string we want to permute (for example "CAT")
2) And the integer k
   - Represents the first k characters that are FIXED at their spots.
   - Nothing left to permute so we print.

```
void RecursivePermute(char str[], int k) {
        int j;

        // Base-case:  All positions are fixed,
        //             Nothing to permute.
        if (k == strlen(str))
                printf("%s\n", str);
```

Recursive Permutation Algorithm

- Recursive Permutation code in detail:
  - Let's use "CAT" as our example
  - Originally we call: RecursivePermute("CAT", 0)
  - Since k == 0, ZERO characters are fixed, so we don't print yet.
    - We move to the else case

```
void RecursivePermute(char str[], int k) {
       // PREVIOUS CODE

       else {
              // Try each letter in spot j
              for (j=k; j<strlen(Str); j++) {
                     // …
              }
       }
```

Recursive Permutation Algorithm

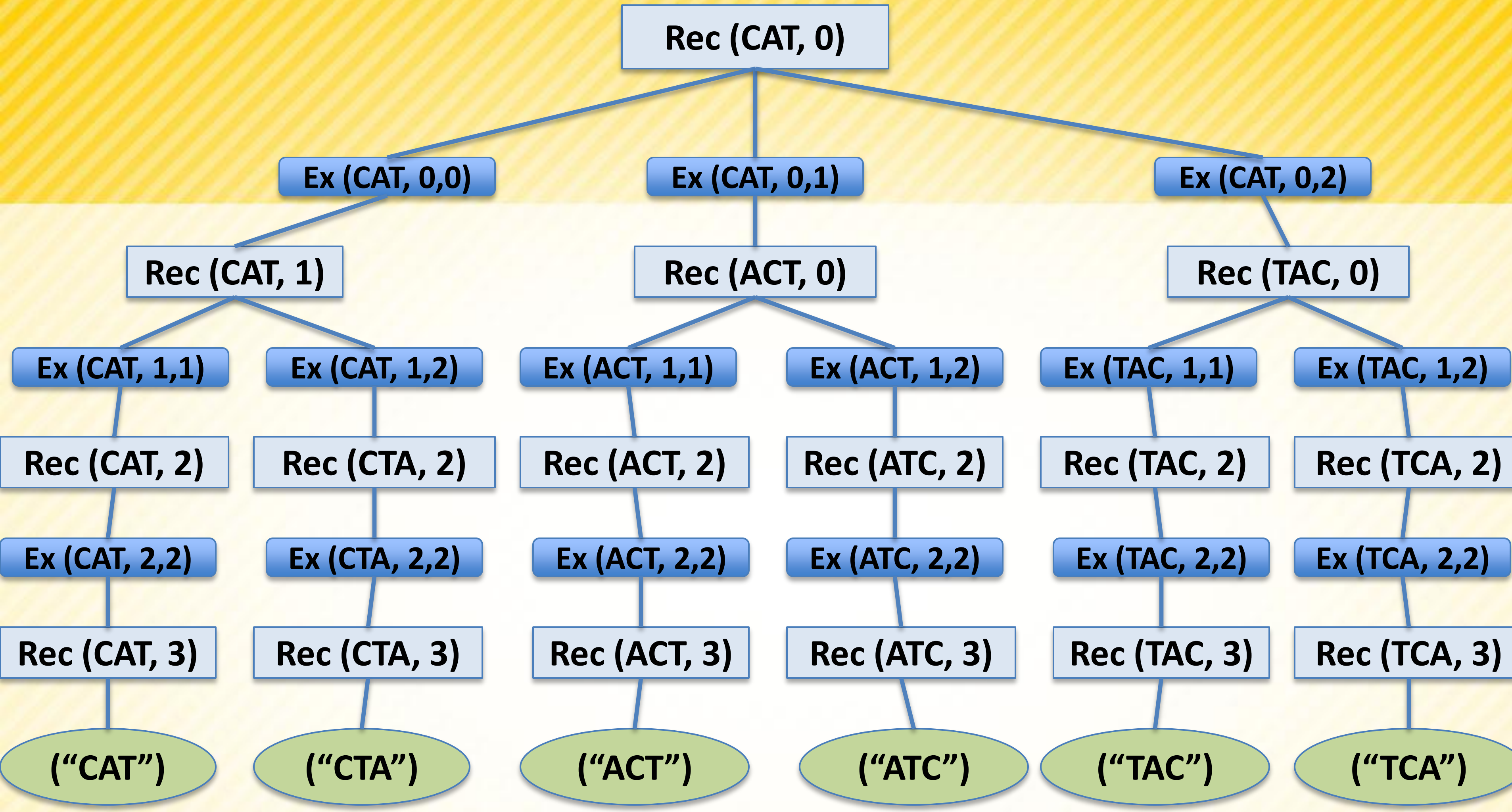
- Recursive Permutation code in detail:
  - ALL other cases (NON-base cases):
    - Call this for loop
    - Iterates the number of times EQUAL to the number of possible characters that can go into index k.

```
// Try each letter in spot j
for (j=k; j<strlen(Str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all perms with spot k fixed
        RecursivePermuite(str, k+1);
        // Put the old char back
        ExchangeCharacters(str, j, k);
}
```

Recursive Permutation Algorithm

- Recursive Permutation code in detail:
  - ALL other cases (NON-base cases):
    - So it would try:
      - Permutations that start with C
      - Permutations that start with A
      - Permutations that start with T

```
// Try each letter in spot j
for (j=k; j<strlen(Str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all perms with spot k fixed
        RecursivePermuite(str, k+1);
        // Put the old char back
        ExchangeCharacters(str, j, k);
}
```




```
for (j=k; j<strlen(Str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```