Lecture 8
Recursion

The Mirrors

Lecture Outline

- Recursion: Basic Idea, Factorial
- Iteration versus Recursion
- How Recursion Works
- Recursion: How to
- More Examples on Recursion
  - Printing a Linked List (in Reverse)
  - Choosing *k* out of *n* Items
  - Tower of Hanoi
  - Fibonacci Numbers
  - Binary Search
  - Permute Strings

Recursion: Basic Idea

- The process of solving a problem with a function that **calls itself** directly or indirectly
  - The solution can be derived from solution of **smaller problem** of the **same type**
- Example: **`Factorial`**
  - **`Factorial(4)`** `= 4 *` **`Factorial(3)`**
- This process can be repeated
  - e.g. `Factorial(3)` can be solved in term of `Factorial(2)`
- Eventually, the problem is so simple that it can solve immediately
  - e.g. `Factorial(0) = 1`
- The solution to the larger problem can then be derived from this …

Recursion: The Main Ingredients

- To formulate a recursive solution:
  - Identify the "simplest" instance

    The **base case(s)** that can be solved *without* recursion
  - Identify "simpler" instances of the same problem

    The **recursive case(s)** that requires recursive calls to solve them
    - Identify how the solution from the simpler problem can help to construct the final result
  - Be sure we are able to reach the "simplest" instance
    - So that we will not get an **infinite recursion**

Example: Factorial

- Let's write a recursive function `factorial(k)` that finds `k!`
  - Base Case:
    - Returns 1 when `k = 0`
    - Corresponding C/C++ code:

```
if (k == 0)
  return 1;
```

  - Recursive Case:
    - Returns `k * (k-1)!`

```
return k * factorial(k-1);
```

Example: Factorial (code)

- Full code for `factorial`:

Max k is 20 before it overflows

```
long factorial(int k) {
  if (k == 0)
    return 1;
  else
    return k * factorial(k-1);
}
```

Base Case:

factorial(0) = 1

Recursive Case:

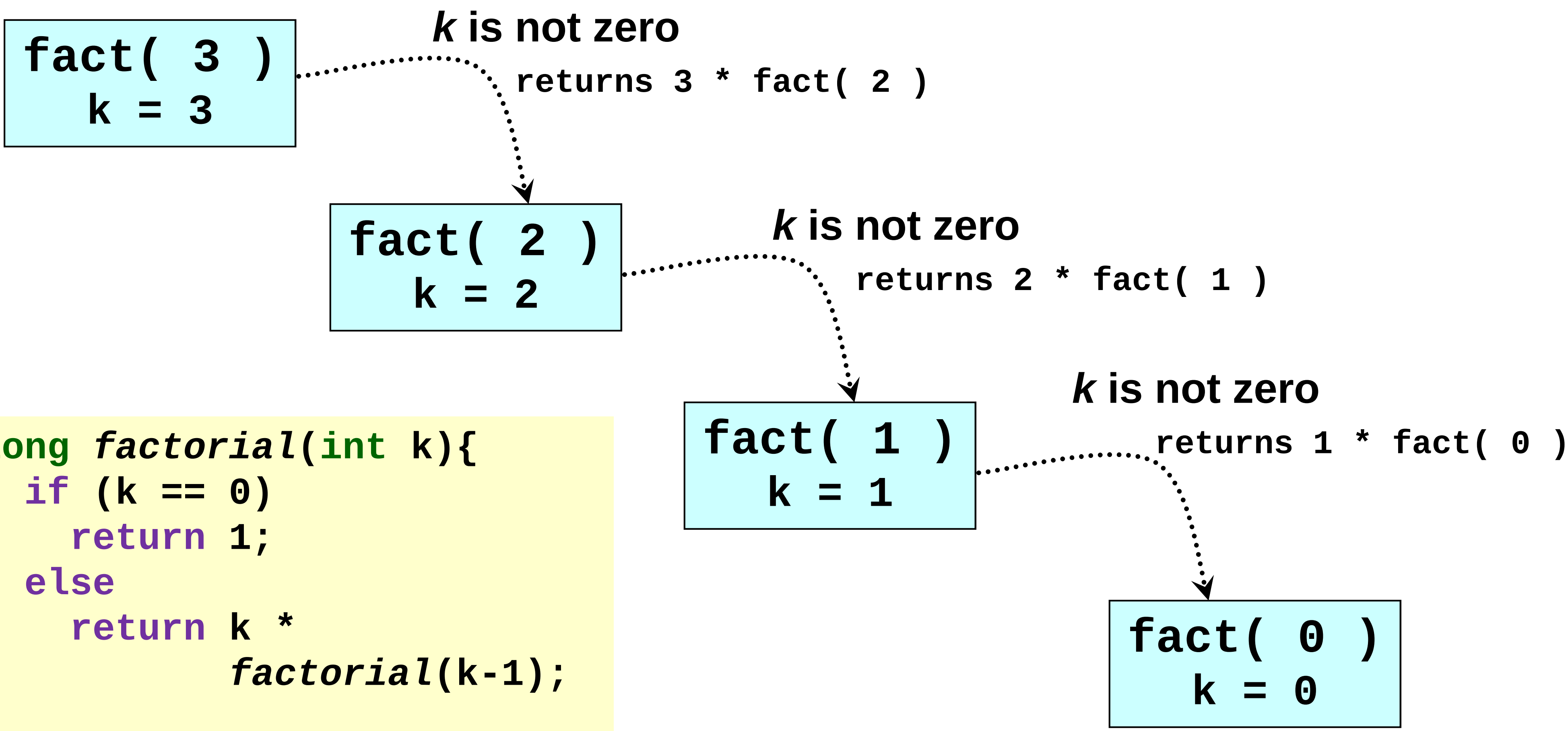
factorial(k) = k * factorial(k–1)

Understanding Recursion

- A recursion always goes through two phases:
  - A **wind-up phase**:
    - When the **base case** is *not* satisfied, i.e. function calls itself
    - This phase carries on **until** we reach the **base case**
  - An **unwind phase**:
    - The recursively called functions return their values to previous "instances" of the function call
      - i.e. the last function returns to its parent (the 2^{nd} last function), then the 2^{nd} last function returns to the 3^{rd} last function, and so on
    - Eventually reaches the very first function, which computes the final value

Factorial: Wind-up Phase

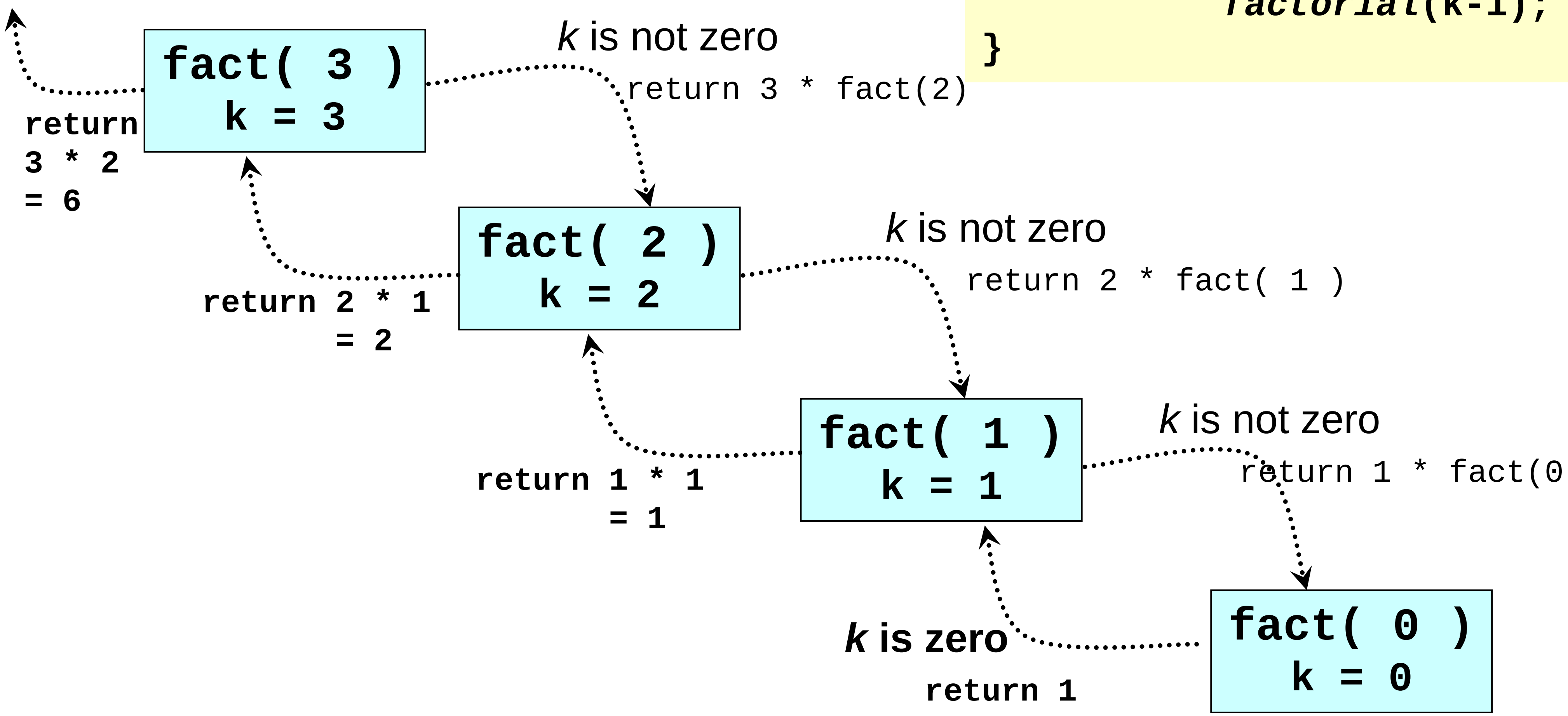
- Let's trace the execution of `factorial(3)` (`factorial` abbreviated as `fact`)




```
long factorial(int k){
  if (k == 0)
    return 1;
  else
    return k *
           factorial(k-1);
}
```

Factorial: Unwind Phase

```
long factorial(int k){
  if (k == 0)
    return 1;
  else
    return k *
         factorial(k-1);
}
```



factorial(3) 6

```
long factorial(int k) {
  if (k == 0)
    return 1;
  else
    return k *  factorial(k-1);
}
```

k: 3

factorial(k-1): 2

factorial(2)

```
long factorial (int k) {
  if (k == 0)
    return 1;
  else
    return k *  factorial(k-1);
}
```

k: 2

factorial(k-1): 1

factorial(1)

```
long factorial (int k) {
  if (k == 0)
    return 1;
  else
    return k *  factorial(k-1);
}
```

k: 1

factorial(k-1): 1

factorial(0)

```
long factorial(int k) {
  if (k == 0)
    return 1;
  else
    …………….
```

k: 0

Recursions vs. Loops

- Many (simple), but not all, recursions essentially accomplish a loop (iterations)
- Recursions are usually much more elegant than its iterative equivalent
  - It is conceptually simple
  - Hence easier to implement
- However iterative version using loops for such recursions is usually faster
- Common practice
  - **If** we convert our recursion to iterative version, we will generally do so

Recursive vs. Iterative Versions

```
long factorial(int k) {
   int j, term;

   term = 1;
   for (j = 2; j <= k; j++)
      term *= j;

   return term;
}
```

Iterative Version

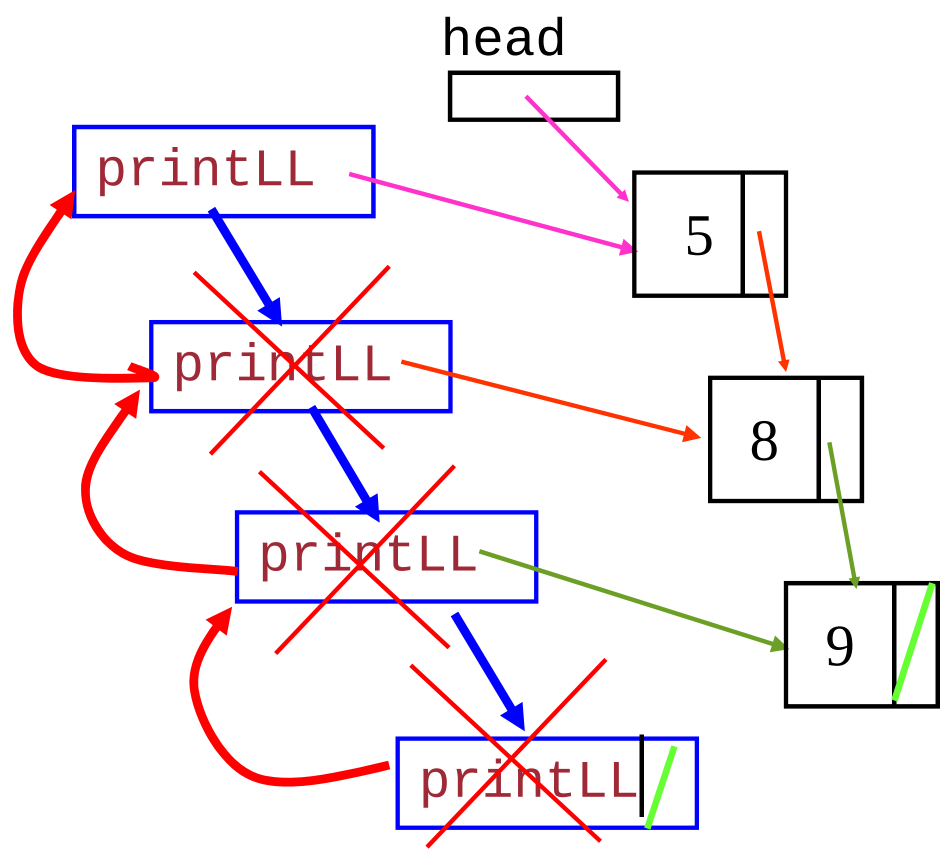
```
long factorial(int k) {
  if (k == 0)
    return 1;
  else
    return k * factorial(k-1);
}
```

Recursive Version

Example: Linked List Printing

- Print out the whole list given the pointer to a ListNode

```
void printLL(ListNode *n){
  if (n != NULL) {
    cout << n->item;
    printLL(n->next);
  }
}
```



Output:

5 8 9

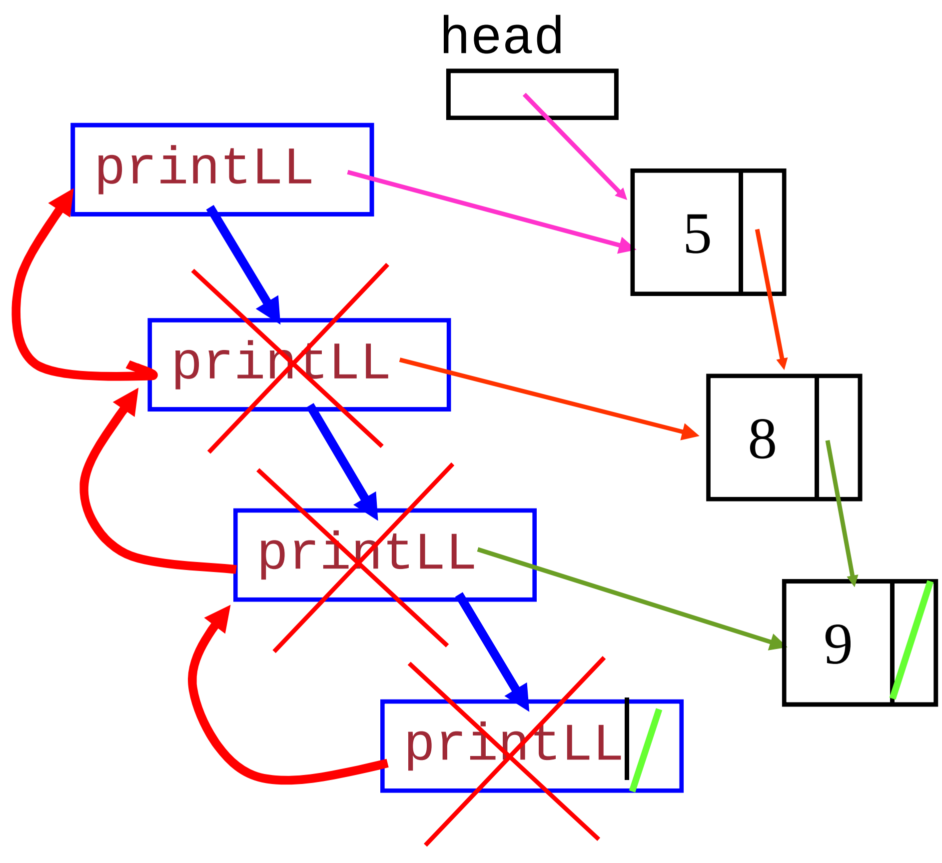
Example: Linked List Printing

- How to print out the whole list in **reverse** order?

```
void printLL(ListNode *n){
  if (n != NULL) {
    printLL(n->next);
    cout << n->item;
  }
}
```

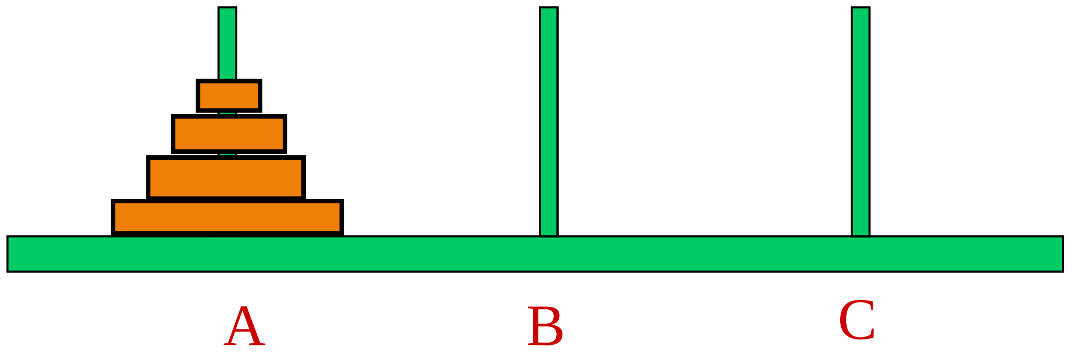
Output:

9 8 5



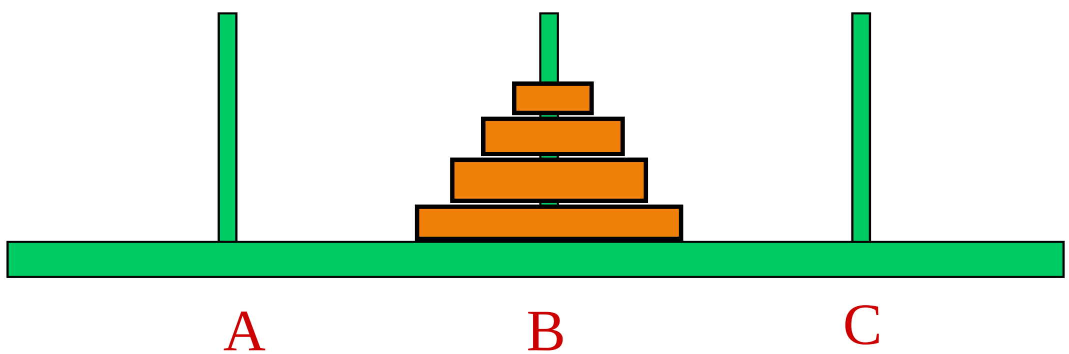
Example: Tower of Hanoi

initial state

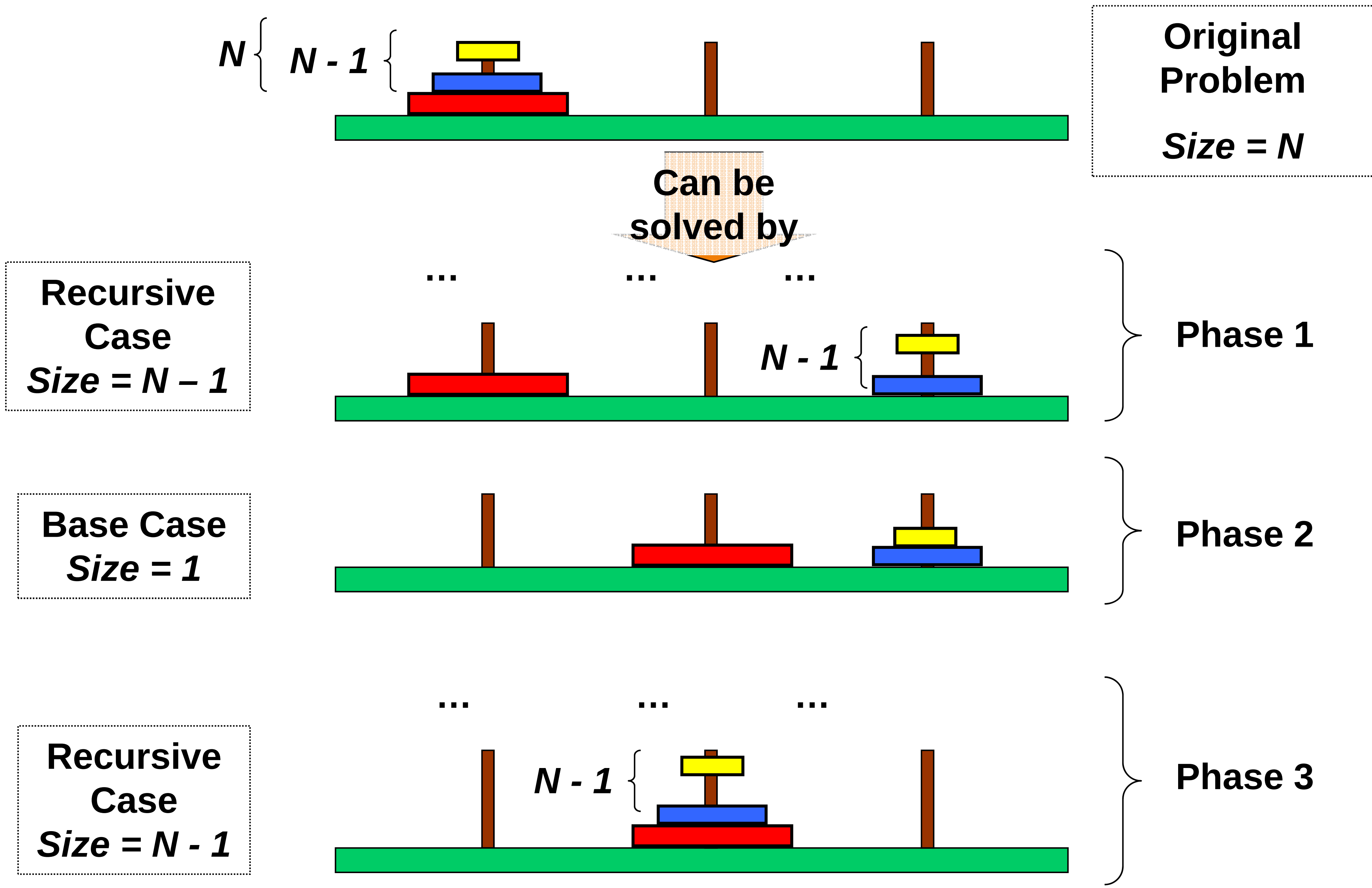


- How do we move all the disks from pole “**A**” to pole “**B**”, using pole “**C**” as temporary storage
  - Move one disk at a time
  - Each disk must not rest on top of a smaller disk

final state



Tower of Hanoi: Recursive Solution



Tower of Hanoi: Solution

```
void tower(int N, char A, char B, char C) {
  if (N == 1)
    move(A, B);
  else {
    tower(N-1, A, C, B);
    move(A, B);
    tower(N-1, C, B, A);
  }
}

void move(char s, char d) {
  cout << "move from " << s << " to " << d << endl;
}
```

Perform the "move". Many implementations. Below is one possibility.

Number of Moves Needed

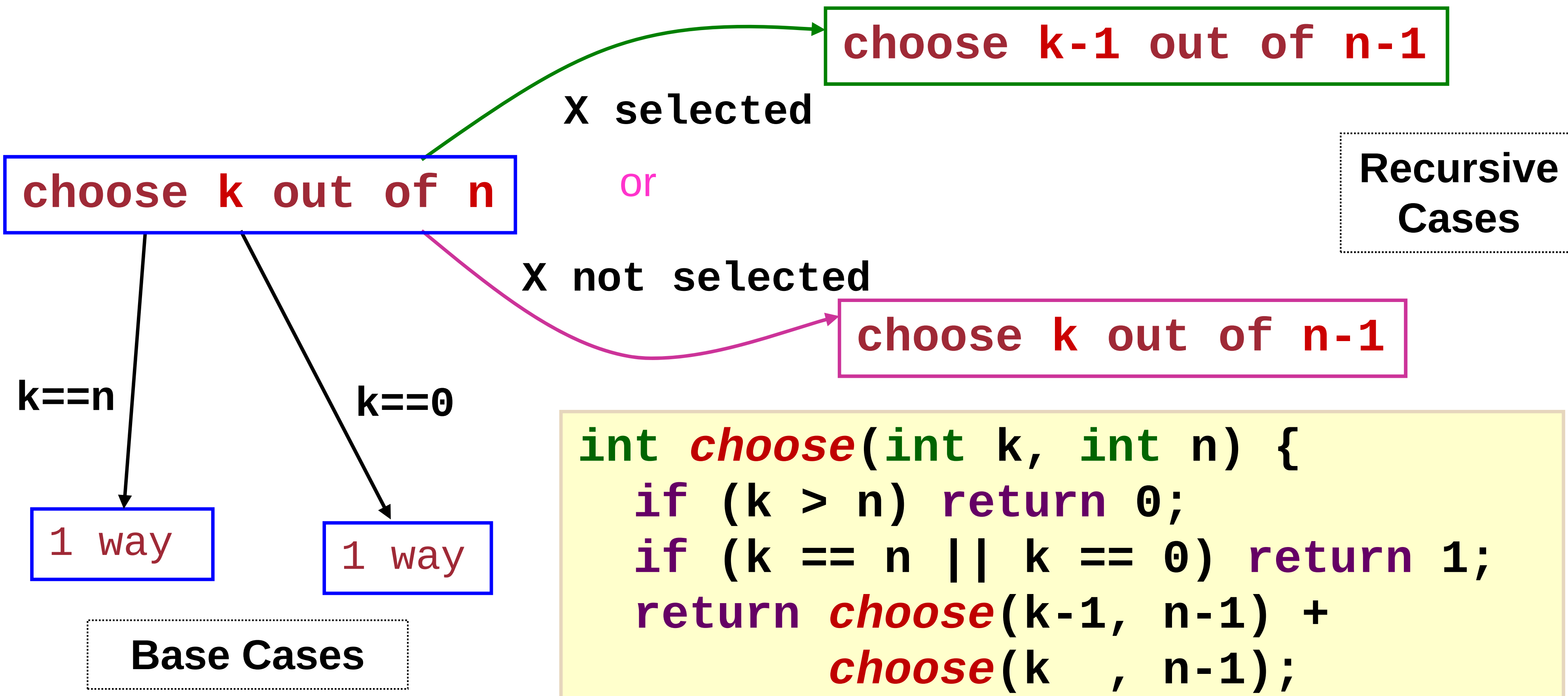
Num of discs, n	Num of moves, f(n)	Time (1 sec per move)
1	1	1 sec
2	3	3 sec
3	3+1+3 = 7	7 sec
4	7+1+7 = 15	15 sec
5	15+1+15 = 31	31 sec
6	31+1+31 = 63	1 min
…	…	…
16	65,536	18 hours
32	4.295 billion	136 years
64	1.8 * 10^10 billion	584 billion years

Note the pattern

$f(n) = 2^n - 1$

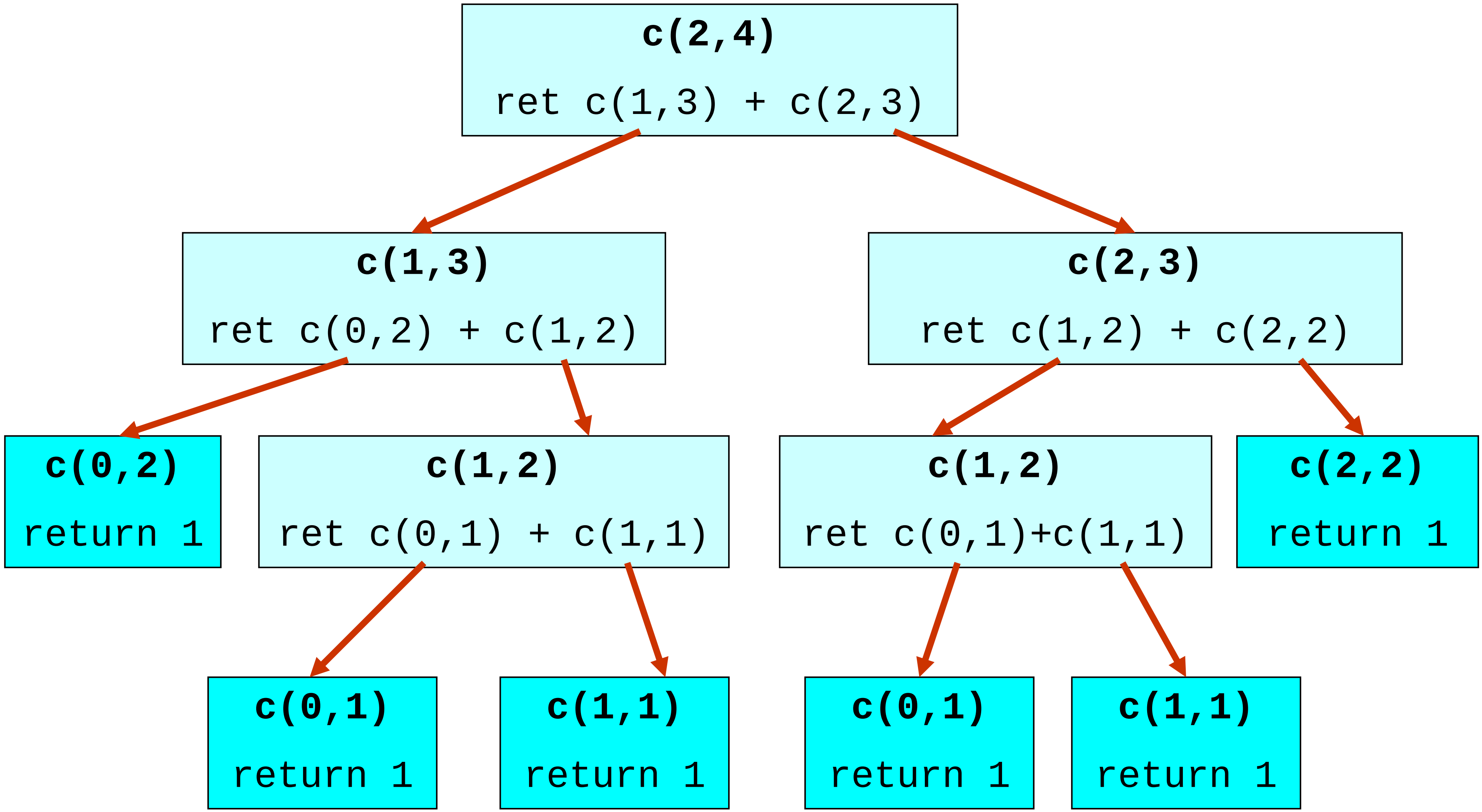
Example: Combinatorial

- How many **ways** can we choose ***k*** items out of ***n*** items?




```
int choose(int k, int n) {
  if (k > n) return 0;
  if (k == n || k == 0) return 1;
  return choose(k-1, n-1) +
         choose(k  , n-1);
}
```

Execution Trace: choose(2, 4)

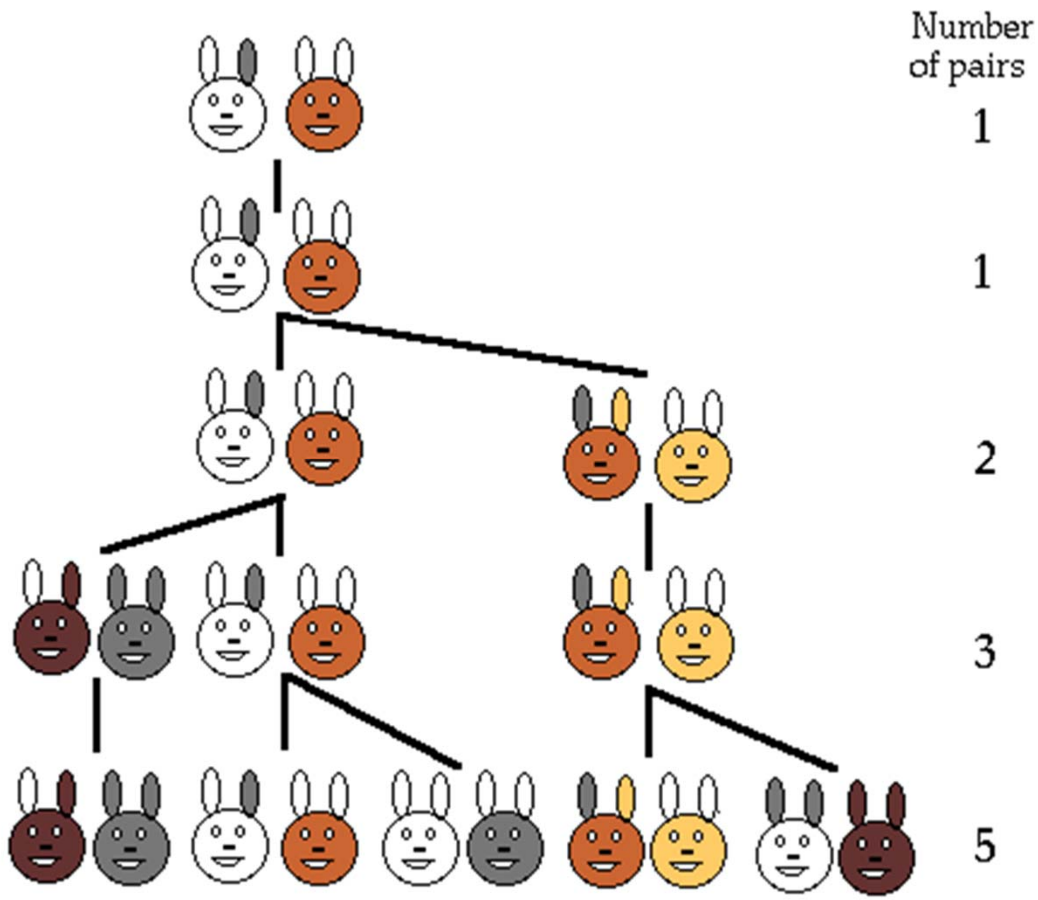


The final answer is the sum of the base cases

Example: Fibonacci Numbers

Rabbits give birth monthly once they are 3 months old and they always conceive a single male-female pair.

Given a pair of male-female rabbits, assuming rabbits never die, how many pairs of rabbits are there after *n* months?



The Fibonacci Series

- `Rabbit(N)`= # pairs of rabbit at Nth month
  - All rabbit pairs in the previous month (N–1)th month stay
    - Rabbits never die
  - Additionally, new rabbit pairs = the total rabbit pairs two months ago (N–2)th month
    - Rabbits give birth at the 3rd month
- Hence:
  - `Rabbit(N) = Rabbit(N–1) + Rabbit(N–2)`
- Special cases:
  - `Rabbit(1) = 1` One pair in the 1st month
  - `Rabbit(2) = 1` Still one pair in the 2nd month
- `Rabbit(N)` is the famous `Fibonacci(N)`

Fibonacci Number: Implementation

```
long fibo(int n) {
   if (n <= 2)
     return 1;
   else
     return fibo(n-1) + fibo(n-2);
}
```

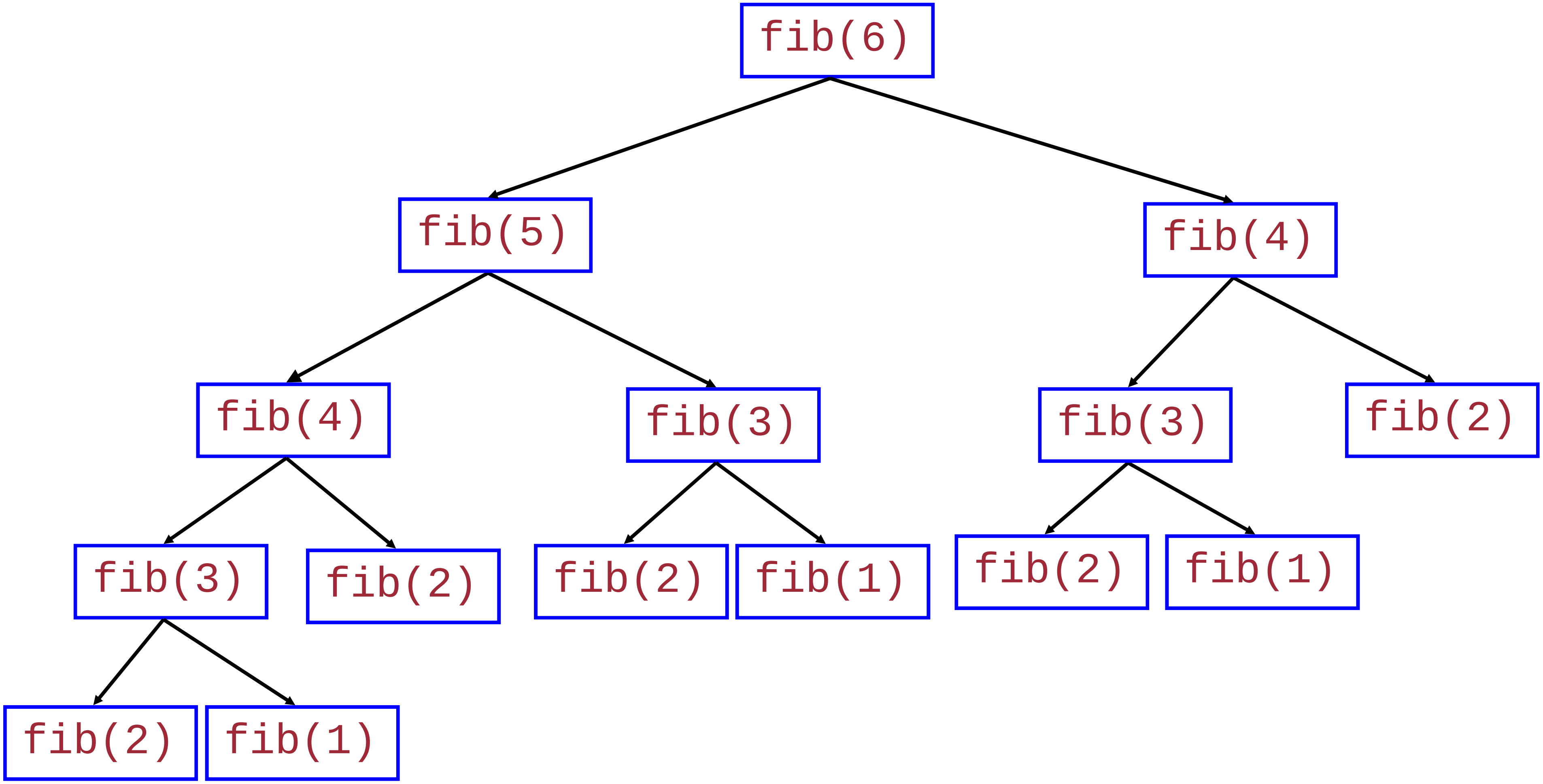
Base Cases:

fibo(1) = 1
fibo(2) = 1

Recursive Case:

fibo(n) = fibo(n–1) + fibo(n–2)

Execution Trace: Fibonacci



- Many duplicate calls
  - The **same computations** are done over and over again!

Fibonacci Number: Iterative Solution

```
long fibo(int n) {
   long cur, prev1 = 1, prev2 = 1, j;

   if (n <= 2)
     return 1;
   else
     for (j = 3; j <= n; j++) {
       cur = prev1 + prev2;
       prev2 = prev1;
       prev1 = cur;
     }
   return cur;
}
```

Iterative Version

- How much time do we need to calculate a particular *fibonacci* number?

Example: Searching in Sorted Array

- Given a **sorted** array **a** of **n** elements and **x**, determine if **x** is in **a**

a =

1	**5**	**6**	**13**	**14**	**19**	**21**	**24**	**32**

x = 15

- How do you reduce the number of checking?
  - Idea: Narrow the search space by half at every iteration until a single element is reached

Binary Search

```
int binarySearch(int a[], int x, int low, int high) {
  if (low > high)   // Base Case 1: item not found
    return -1;

  int mid = (low+high) / 2;

  if (x > a[mid])
    return binarySearch(a, x, mid+1, high);
  else if (x < a[mid])
    return binarySearch(a, x, low, mid-1);
  else
    return mid;    // Base Case 2: item found
}
```

Example: Find all Permutations of a String

- Given a word, say *east*, the program should print all 24 permutations (anagrams), including *eats*, *etas*, *teas*, and non-words like *tsae*

- One idea to generate all permutations (other ways exist)
  - Given *east*, we place the first character, i.e. *e*, in front of all 6 permutations of the other 3 characters *ast* — *ast*, *ats*, *sat*, *sta*, *tas*, and *tsa* — to arrive at *east*, *eats*, *esat*, *esta*, *etas*, and *etsa*, then
  - We place the second character, i.e. *a*, in front of all 6 permutations of *est*, then
  - We do the same for characters *s* and *t*
  - Thus, there will be 4 (the size of the word) recursive calls to display all permutations of a four-letter word

- Of course, when we're going through the permutations of 3-character string, e.g. *ast*, we would follow the same procedure

Example: Find all Permutations of a String

```
void permuteString(string beginningString,
                   string endingString) {
  if (endingString.length() <= 1)
    cout << beginningString << endingString << endl;
  else
    for (int i = 0; i < endingString.length(); i++) {
      string newString = endingString.substr(0, i) +
                         endingString.substr(i+1);
      permuteString(beginningString + endingString[i],
                    newString);
   }
}
```

- Start by calling **`permutateString("", "east");`**

Summary

- Recursion is not just a way of programming, it is also a powerful approach to problem solving and formulating a solution
- A recursive function has base cases and recursive cases
- Relationship between recursion and stack
- Watch out for duplicate computations!

VisuAlgo Recursion Tree Visualization

- http://visualgo.net/recursion
- Accepts any valid (JavaScript) recursive function with starting input parameter
- Green vertices: base cases
- Blue vertices: Repeated cases
- Red text: Return values