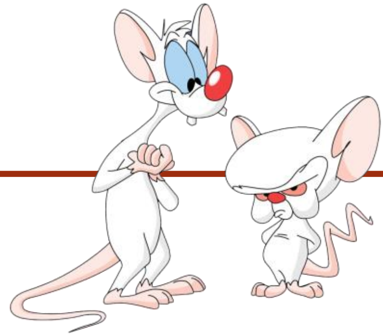
COMP 250

INTRODUCTION TO COMPUTER SCIENCE

Week 9-2 : Recursion 1

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Recursive algorithms

EXAMPLE

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.print(“Go!”);
    } else {
        System.out.print(n + “ ”);
        countdown(n-1);
    }
}
```


- What prints if we call `countdown(3)`?
  - `3 2 1 Go!`

EXAMPLE – EXECUTION

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.print(“Go!”);
    }   else {
        System.out.print(n + “ ”);
        countdown(n-1);
    }
}
```


Execution of `countdown(3)`.

- The execution of `countdown` starts with `n==3`. Since it is not 0, `3` is printed and `countdown` is called with input 2
  - The execution of `countdown` starts with `n==2`. It is not 0, thus `2` is printed and `countdown` is called with input 1.
    - The execution of `countdown` starts with `n==1`. Since it is not 0, `1` is printed and `countdown` is called with input 0.
      - The execution of `countdown` starts with `n==0`. Since n is 0, `Go!` is printed and the execution ends.
    - The execution of `countdown(1)` ends.
  - The execution of `countdown(2)` ends.
- The execution of `countdown(3)` ends and we are back in main.

RECURSIVE – DEFINITION

Recursive functions/methods consists of the following

- **Base Case(s)**: one (or a finite number) of terminating scenario that does not use recursion to produce an answer.
- **Recursive or Inductive step(s)**: rules that determine how to produce an answer from simpler cases.

BASE CASE

Note that if there is no base case in a recursive method, or if the base case is never reached, the execution will never end.


```
public static void forever (int n) {
   forever(n);
}
```

COMING UP

Several examples of algorithms that can be implemented recursively:

- Factorial function
- Fibonacci numbers
- reverseList
- sortList
- towerOfHanoi

EXAMPLE 1 – FACTORIAL

The factorial of a number is defined as follows:

$0! = 1$

$1! = 1$

$2! = 1 * 2 = 2$

$3! = 3 * 2 * 1 = 6$

...

$n! = n * (n-1) * (n-2) * (n-3) * \ldots * 1$

FACTORIAL: RECURSIVE DEFINITION

- Notice that:

$$n! = n * (n-1) * (n-2) * (n-3) * \ldots * 1$$
$$= n * (n-1)!$$

- Thus, the following definition completely determines the factorial:

Base case: $0! = 1$

Recursive step: $n! = n * (n-1)!$

FACTORIAL (ITERATIVE)


```
public static int factorial (int n) {
    int result = 1;
    for(int i=2; i<=n; i++) {
        result = result * i;
    }
    return result;
}
```

FACTORIAL (RECURSIVE)

Let's use its recursive definition to write a method that computes the factorial function:


```
public static int factorial (int n) {
   if (n == 0) {
      return 1;
   }
   return n * factorial(n-1);
}
```

Base case

Induction step

FACTORIAL: AN EXAMPLE

- What happens when the method call `factorial(4)` is executed?


```
public static int factorial (int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Factorial(4) → returns 24

Factorial(3) → returns 6

Factorial(2) → returns 2

Factorial(1) → returns 1

Factorial(0) → returns 1

CORRECTNESS

Claim: the recursive `factorial(n)` algorithm returns $n!$.

Proof (by mathematical induction):

- Base case: `factorial(0)` returns 1.
- Induction step:
  - IH: Assume `factorial(k)` returns $k!$ when $k \geq 1$
  - To prove: `factorial(k+1)` returns $(k+1)!$

  `factorial(k+1)` returns `factorial(k)` $* (k+1)$

  $= k! * (k+1)$, by IH

  $= (k+1)!$

EXAMPLE 2 – FIBONACCI NUMBERS

- Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

- Base cases:

$$f_1 = 1$$

$$f_2 = 1$$

- Recursive/Inductive Step:

$$f_n = f_{n-1} + f_{n-2}$$

FIBONACCI (ITERATIVE)

```
public static int fibonacci(int n) {
   if(n==0 || n==1) {
      return 1;
   }
   fib0 = 1;
   fib1 = 1;
   for(int i=2; i<=n; i++) {
      fib2 = fib0 + fib1;
      fib0 = fib1;
      fib1 = fib2;
   }
   return fib2;
}
```

FIBONACCI (RECURSIVE)

```
public static int fibonacci (int n) {
   if(n==0 || n==1) {
      return 1;
   }
   return fibonacci(n-1)+fibonacci(n-2);
}
```


This is much simpler to express than the iterative version.

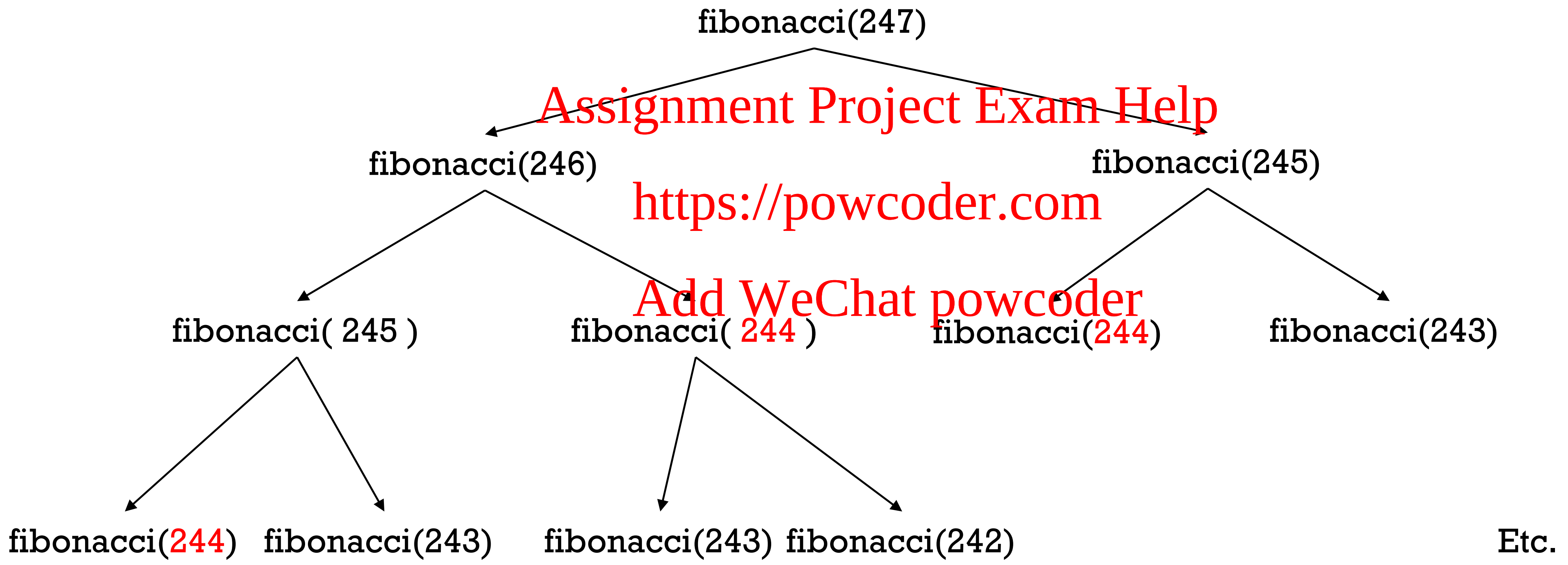
CORRECTNESS

Claim: the recursive Fibonacci algorithm is correct.

Proof(sketch): (by strong mathematical induction)

- Induction step:
  - IH: Let $k \geq 0$, Assume `fibonacci(i)` returns f_i for every $0 \leq i < k$
  - To prove: `fibonacci(k)` returns f_k

However, the recursive Fibonacci algorithm is very inefficient.
It computes the same quantity many times, for example:



EXAMPLE 3: REVERSING A LIST

input $\{a, b, c, d, e, f, g, h\}$

output $\{h, g, f, e, d, c, b, a\}$

EXAMPLE 3: REVERSING A LIST

input $\{a, b, c, d, e, f, g, h\}$

output $\{h, g, f, e, d, c, b, a\}$

Idea of recursion:

$a \quad \{b, c, d, e, f, g, h\}$

$\{h, g, f, e, d, c, b\} \quad a$

REVERSING A LIST (RECURSIVE)

```
public static void reverse(List list) {
    if(list.size()==1) {
        return;
    }
    firstElement = list.remove(0); // remove first element
    reverse(list); // now the list has n-1 elements
    list.add(firstElement); // appends at the end of the list
}
```

EXAMPLE 4 – SORTING A LIST (RECURSIVE)

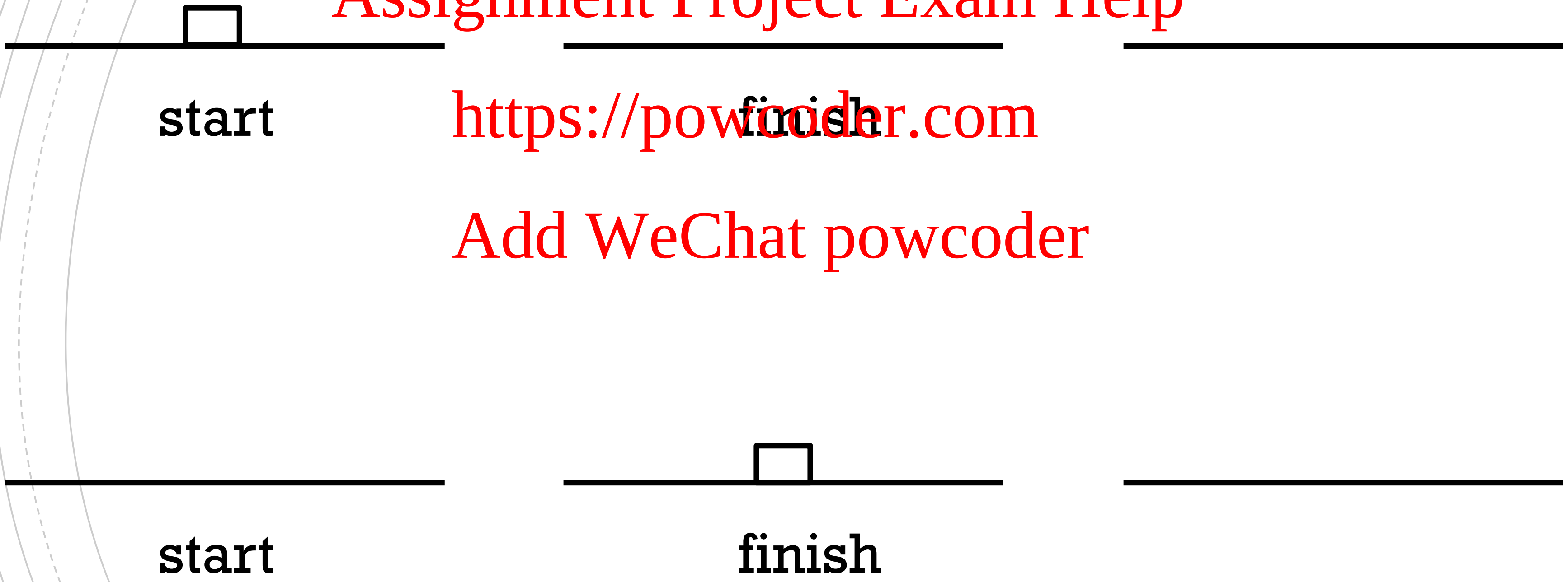

```
public static void sort(List list) {
   if(list.size()==1) {
      return;
   }
   minElement = removeMinElement(list);
   sort(list); // now the list has n-1 elements
   list.add(0, minElement); // insert at the beginning of list
}
```

EXAMPLE 5 – TOWER OF HANOI

Problem: Move n disks from start tower to finish tower such that:

- move one disk at a time
- you can have a smaller disk on top of bigger disk (but you can't have a bigger disk onto a smaller disk)

EXAMPLE - n=1



EXAMPLE - n=2

start finish

from A to C

start finish

EXAMPLE - n=2

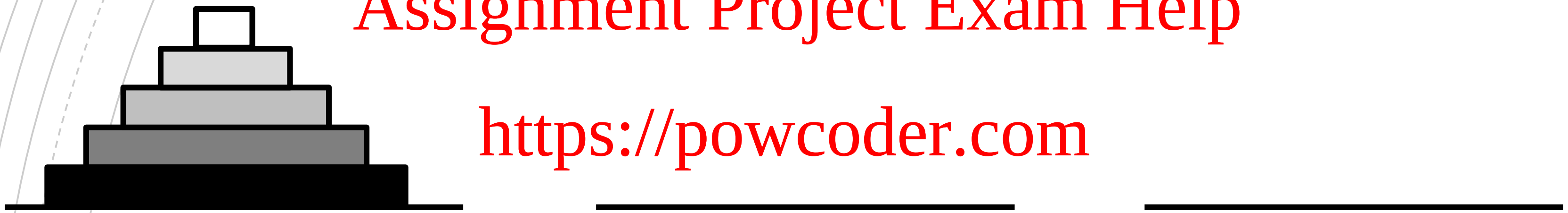
from A to B

start finish

from C to B

start finish

HOW SHOULD WE MOVE 5 DISKS FROM A TO B?



- Let's think about it recursively!

IDEA!

Somehow move 4 disks from A to C

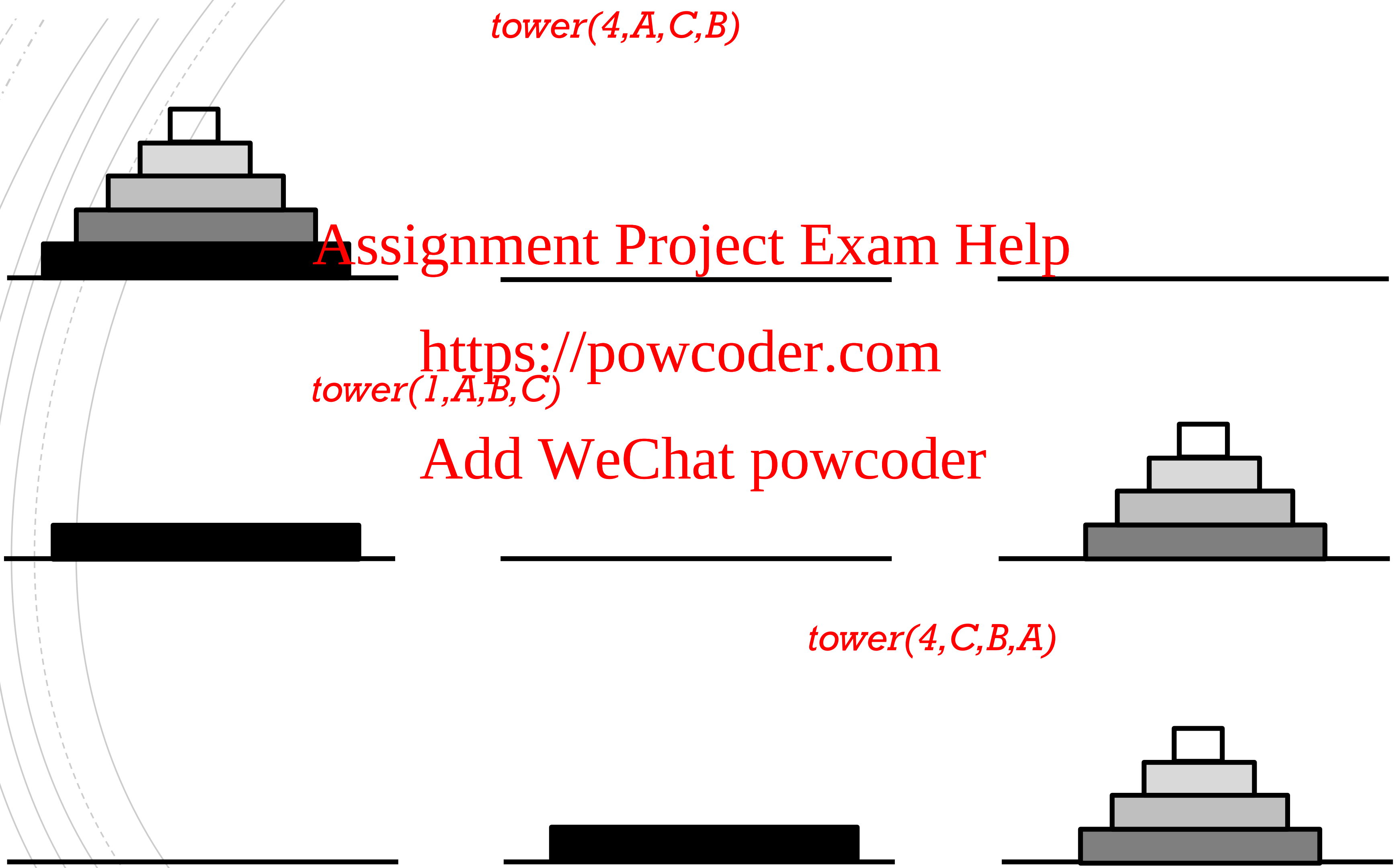
move 1 disk from A to B

Somehow move 4 disks from C to B

ALGORITHM

```
tower(n, start, finish, other) { // e.g. tower(5,A,B,C)
   if(n==1) {
      move from start to finish.
   } else {
      tower(n-1, start, other, finish)
      tower(1, start, finish, other)
      tower(n-1, other, finish, start)
   }
}
```

EXAMPLE



CORRECTNESS

Claim: the tower( ) algorithm is correct, namely it moves the blocks from start to finish without breaking the two rules (one at a time, and can't put bigger one onto smaller one).

<u>Proof: (**sketch**)</u>

- Base case: `tower(1,*,*,*)` is correct.
- Induction step:
  - for any k > 1, assume `tower(k,*,*,*)` is correct
  - Prove `tower(k+1,*,*,*)` is correct.

HOW MANY MOVES?

tower(1, start, finish, other)

move from

start to finish

Answer: 1

HOW MANY MOVES?

tower(2, start, finish, other)

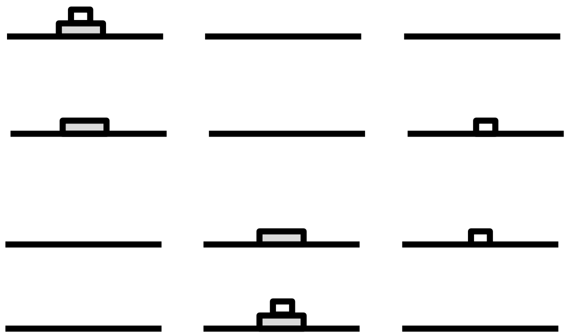
tower(1, start, other, finish)

move

tower(1, other, finish, start)

move

move



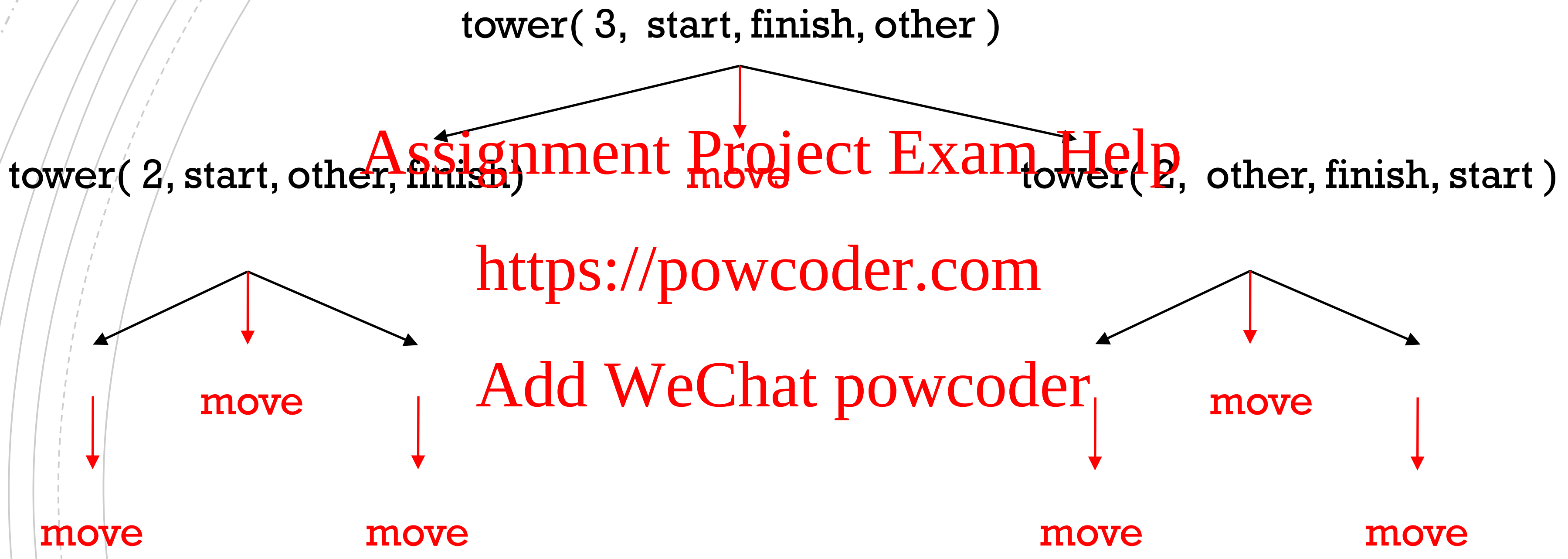
move from A to C

move from A to B

move from C to B

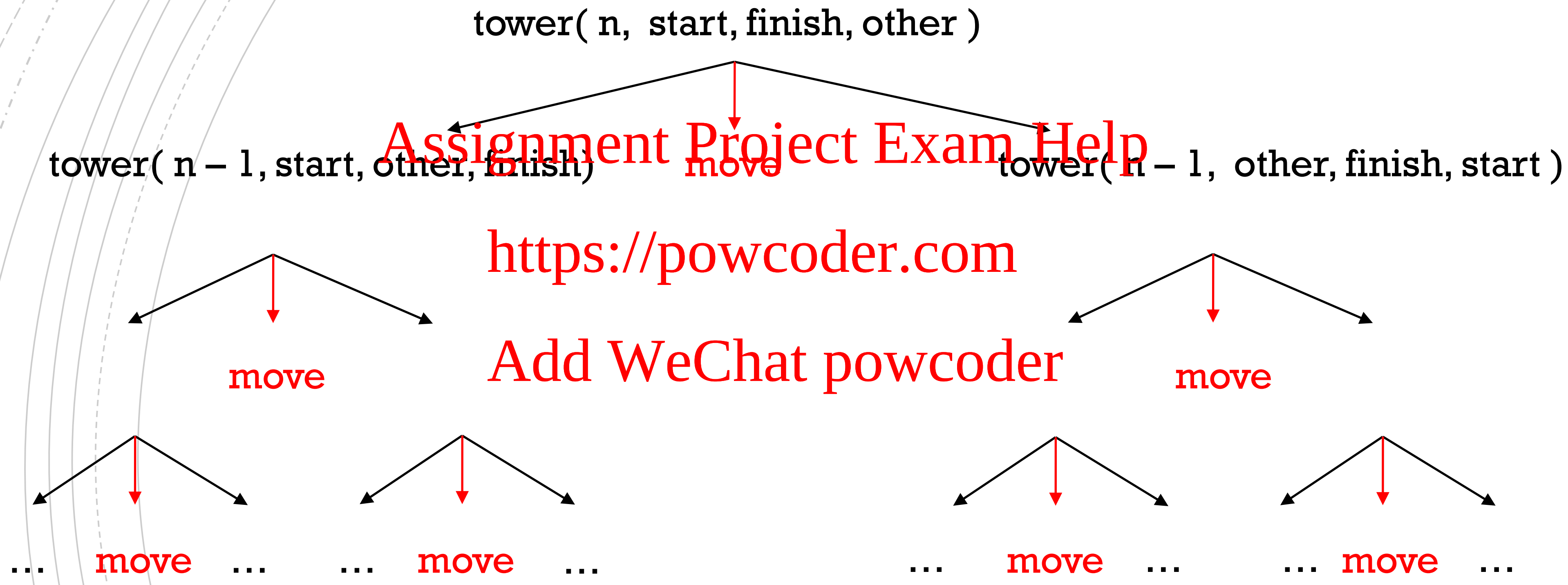
Answer: 1 + 2

HOW MANY MOVES?



Answer: $1 + 2 + 4 = 2^0 + 2^1 + 2^2$

HOW MANY MOVES?



Answer: $1 + 2 + 4 + \ldots + 2^{n-1} = 2^{n} - 1$

RECURSION AND ITERATION

- Recursion and iteration (loops) are equally expressive.

  - Anything recursion can do, iteration and do

  - Anything iteration can do, recursion can do

RECURSION VS ITERATION

- Which one to use?
  - Use the one is easier to think in terms of, for a specific problem.
  - For simple cases, iteration is usually easier and faster.
  - For complex cases, recursion is often more elegant and simpler to code.
  - It is important to remember that when using one or the other, this decision might impact the performance of your program.

RECURSIVE DATA STRUCTURE

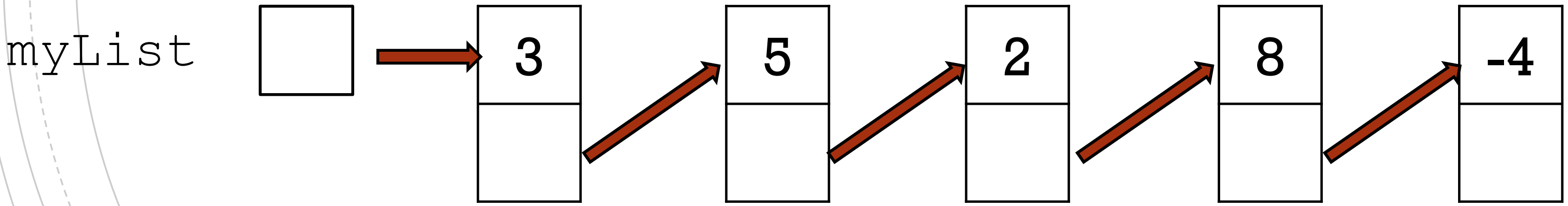
- We can recursively defined also data structures.

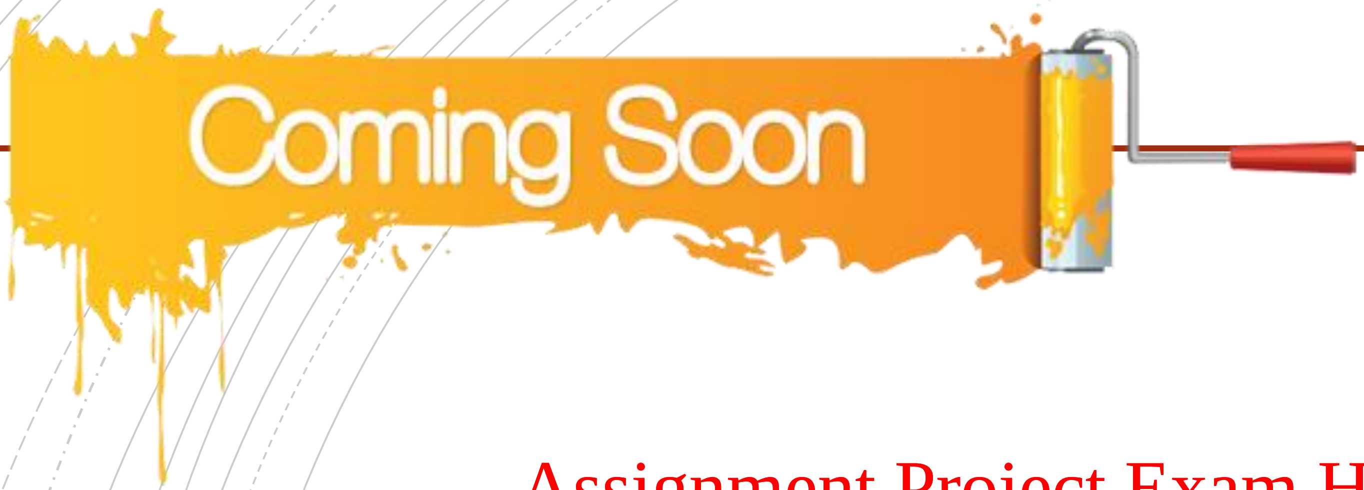
- Let's consider arrays and let's think how we can recursively defined a list of items.

LINKEDLIST

- **LinkedList<E> class :**

```
private E val;
private LinkedList<E> next;
```



In the next videos:

- Binary Search