Recursion

- Explaining Recursion
- Recursion Trees
- Recursion and Memory
- Recursion vs. Iteration
- Different Types of Recursion
- Example Problems and Applications

What Algorithm Is This?

Algorithm: SomeAlgo(A[1..n])

```
for i = n to 2 do
    for j = 2 to i do
        if A[j] < A[j−1] then
            t = A[j];
            A[j] = A[j−1];
            A[j−1] = t;
```

Recursion: What You Should Already Know

- Function / method which calls itself (directly or indirectly)
- Recursive case and base case
- Can be rewritten as iteration (and vice-versa)

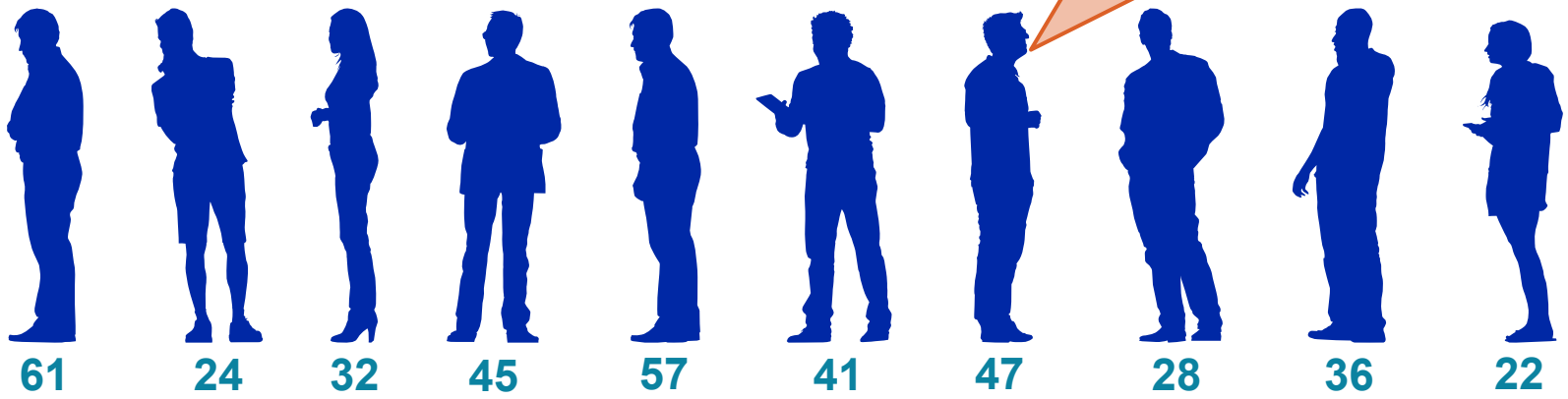
- Main idea: solve a smaller problem which is an exact copy of the bigger problem but smaller

Explaining Recursion

Assume you want to explain recursion to your 12 year old cousin.

How would you do this?

Consider a queue of people as shown below.
How can we find out the total age of all the people in the queue?



Explaining Recursion: Iterative vs. Recursive Solution

iterative approach:
go from person to person and add together their ages

Algorithm: GetTotalAgeIter(peopleQueue)

```
totalAge = 0
for k = 1 to length(peopleQueue) do
    totalAge = totalAge + age of k^th person
```

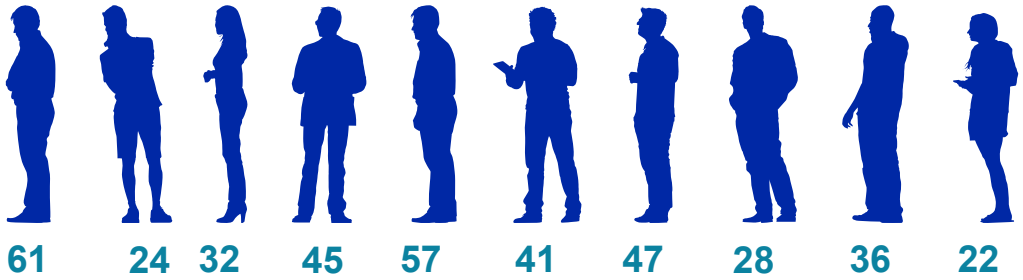
recursive approach:
ask person behind you to find out using the same strategy

Algorithm: GetTotalAgeRec(peopleQueue)

```
if no person standing behind you then
    tell person before you your own age
else
    ask person behind you for age
    tell answer to person before you
```

Explaining Recursion

Use a C array to represent ages of people in queue; use C language to implement the solution:

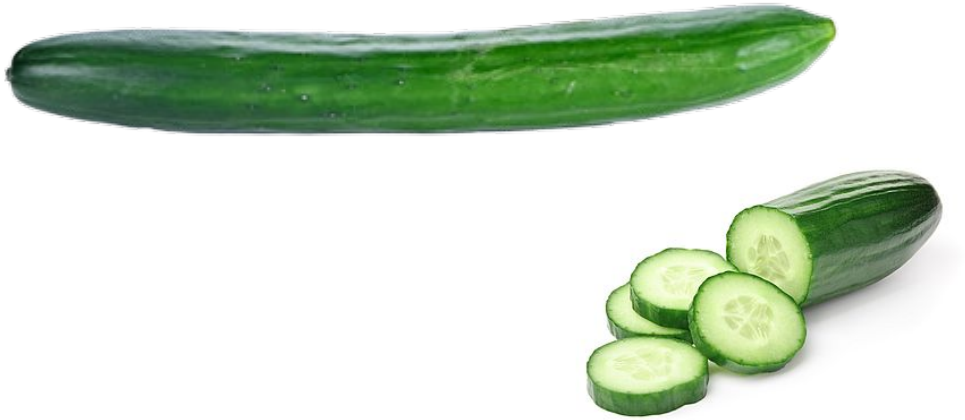


$$sum(A[1..k]) = \begin{cases} A[k] & \text{if k is the last position} \\ sum(A[1..k-1]) + A[k] & \text{otherwise} \end{cases}$$

arr =	61	24	32	45	57	41	47	28	36	22
	0	1	2	3	4	5	6	7	8	9

Explaining Recursion: How To Make Cucumber Salad

- How do cut a cucumber to make a salad iteratively and recursively?
- How do you eat a plate of soup recursively?



Cucumber Salad and Soup: Pseudocode Algorithms

Algorithm: cut(cucumber)

if less than 1 cm of the cucumber is left **then**
    stop, you're done

else
    remove a 1 cm slice from the cucumber
    cut(remaining cucumber)

Algorithm: eat(soup)

if nothing of the soup is left **then**
    stop, you're done

else
    eat as much of the soup as fits on spoon
    eat(remaining soup)

Recursion: «Recursive Leap of Faith»

How can you transport 1000 elephants to the moon?

→ If all elephants already are on the moon, we're done. Else, ship one elephant to the moon first and then use the some approach on the remaining 999.

This kind of logic applied in recursive solutions oftentimes can seem a bit «magical». One just has a solution for a very small (sometimes even trivial) problem and then just calls the same approach on a reduced copy of the complete problem which then will «somehow» solve the problem.

This part of constructing a recursive solution is therefore sometimes called the «Recursive Leap of Faith».

Exercise Recursion: Average of an Integer Array

Calculate the average of the elements in an array of integers recursively.

Exercise Recursion: Average of an Integer Array

By definition, the average is the sum of all elements divided by the number of the elements. Conversely, the sum can be expressed as the average multiplied with the number of elements:

$$\text{avg}(A[1..n]) = \frac{\text{sum}(A[1..n])}{n} \quad \Rightarrow \quad \text{sum}(A[1..n]) = \text{avg}(A[1..n]) \cdot n$$

Further, we can replace the sum of an array by adding the very last element to the sum of the all the previous elements of the array. The latter can then be expressed in terms of its average as shown above:

$$\text{sum}(A[1..n]) = \text{sum}(A[1..n-1]) + A[n] = \big(\text{avg}(A[1..n-1]) \cdot (n-1)\big) + A[n]$$

Hence, we can write the following recursive formulation for the average:

$$\text{avg}(A[1..n]) = \begin{cases} A[1] & \text{if } n = 1 \\ \dfrac{\big(\text{avg}(A[1..n-1]) \cdot (n-1)\big) + A[n]}{n} & \text{otherwise} \end{cases}$$

Exercise Recursion: Average of an Integer Array

```
1  double average(double arr[], int n) {
2       if (n == 1) {
3            return arr[1];
4       } else {
5            return (average(arr, n-1) * (n-1) + arr[n-1]) / n;
6       }
7  }
```

base case

recursive call

Recursion: Code Example I

Consider the following recursive function:

```
int recFun(int n) {
      if (n < 10) {
            return n;
      } else {
            int a = n / 10;
            int b = n % 10;
            return recFun(a + b);
      }
}
```

What is the result of the call `recFun(648)`?

A: **8**
B: **9**
C: **54**
D: **72**
E: **648**

Result is B: 9.

The recursion tree looks as follows:

recFun(648)
|
recFun(72)
|
recFun(9)

Recursion: Code Example II

Consider the following code snippet. What will be the output?

```c
#include <stdio.h>

void whatDoesItDo(int n) {
    if (n > 0) {
        int a = 1;
        int b = 2;
        int c = a + b;
        whatDoesItDo(c);
    }
    else {
        printf("world!\n");
    }
}
```

```c
int main() {
    printf("Hello ");
    int x = 42;
    whatDoesItDo(x);
    return 0;
}
```

→ **Segmentation fault: stack overflow**. The base case of the recursive function can never be reached. Note that the «steering variable» n of the recursion will never be changed and remain 3 if it is bigger than zero in the first call of the function. Therefore the memory will be filled with return addresses until has been used completely – at which time the program will crash.

Recursion Trees: Example

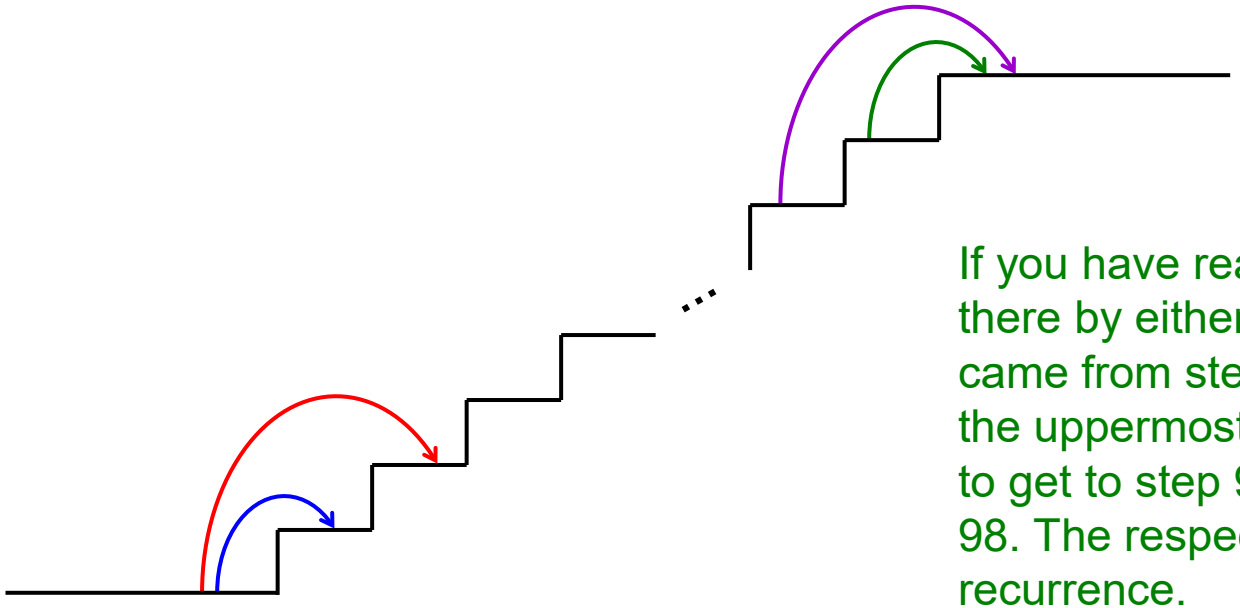
What will be the result of a call of recFun(3, 0)?

```
int recFun(int depth, int sum) {
  if (depth <= 0) {
    sum = sum + 5;
    return sum;
  }
  else {
    int t1 = recFun(depth - 1, sum + 2);
    int t2 = recFun(depth - 2, sum + 3);
    return t1 + t2;
  }
}
```

Optimal substructure example 1: number of ways to climb up stairs

Consider a staircase with N = 100 steps. Assume, a person can climb up one or two stairs at a time. After each step, he/she decides whether to take one or two stairs in the next step.

How many different ways are there to climb up the N = 100 stairs?



If you have reached the uppermost step (step 100), you got there by either taking one stair or two stairs, so you either came from step 98 or step 99. The number of ways to get to the uppermost step is therefore equal to the number of ways to get to step 99 the plus the number of ways to get to step 98. The respective recurrence is equivalent to the Fibonacci recurrence.

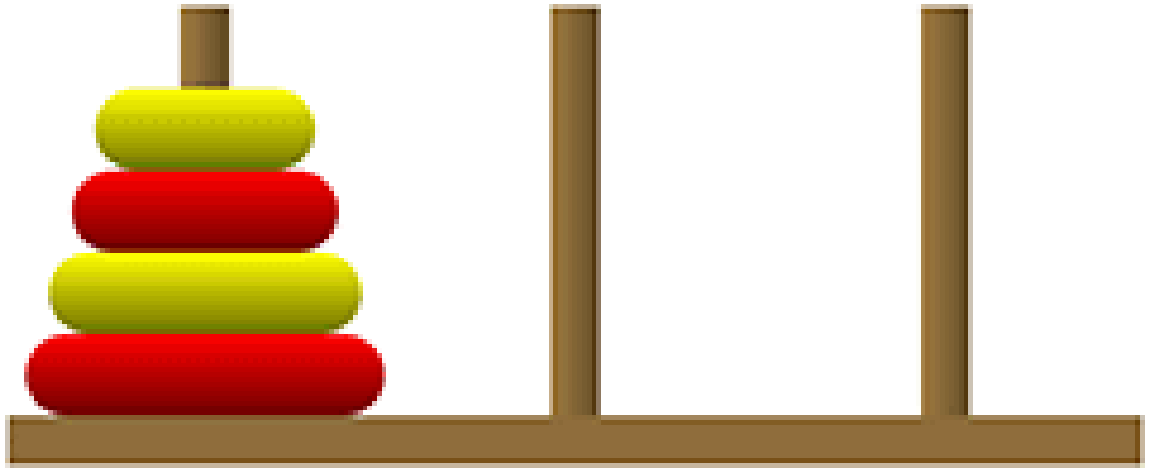
Recursion: Comprehension Question

What is the difference between an infinite loop and an infinite recursion?

Recursion: Comprehension Question

Can there be a meaningful recursive function which has return type `void` and no arguments, e.g. a function with signature `void myRecFun(void)`? Explain your answer.

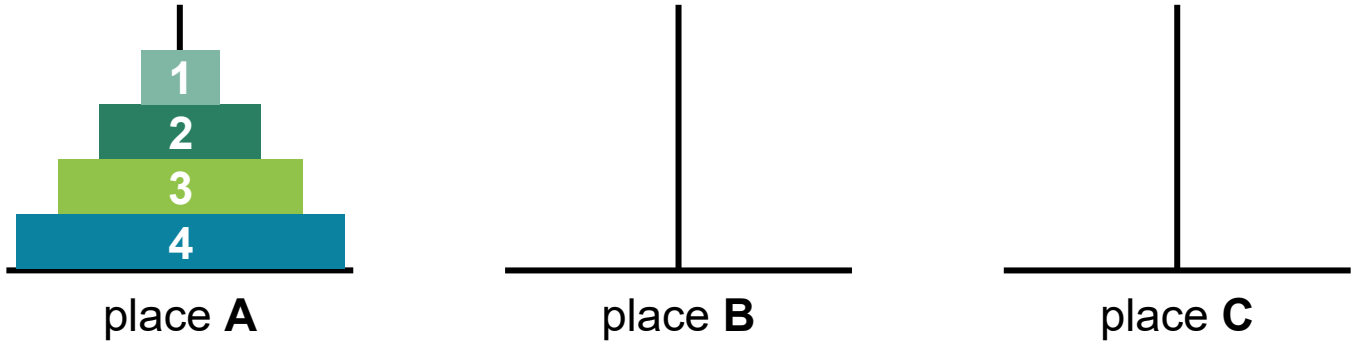
Towers of Hanoi



(from http://larc.unt.edu/ian/TowersOfHanoi/4-256.gif)

Towers of Hanoi: Example with Four Disks

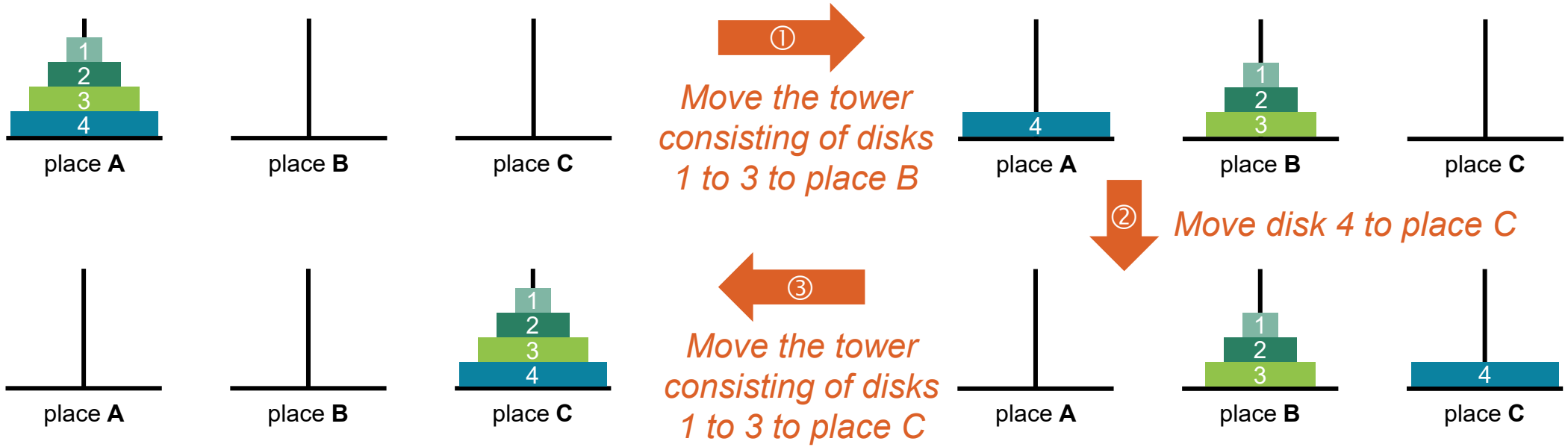
Let's number the disks of the tower from 1 to N, starting with the smallest disk and let's denote the three places as A, B and C. Thus the starting configuration looks like this:



We now are given the task to move the four disks from place A to place C without having a disk with a higher number above a disk with a lower number at any point in time.

Towers of Hanoi: Example with Four Disks

The problem of shifting four disks from place A to place C can be solved in three steps as follows:



So far so good, but now we need a way to move a tower of three disks from one position to another…

How can we solve this? → By applying the same strategy recursively!

Towers of Hanoi: Generalized Recursive Approach

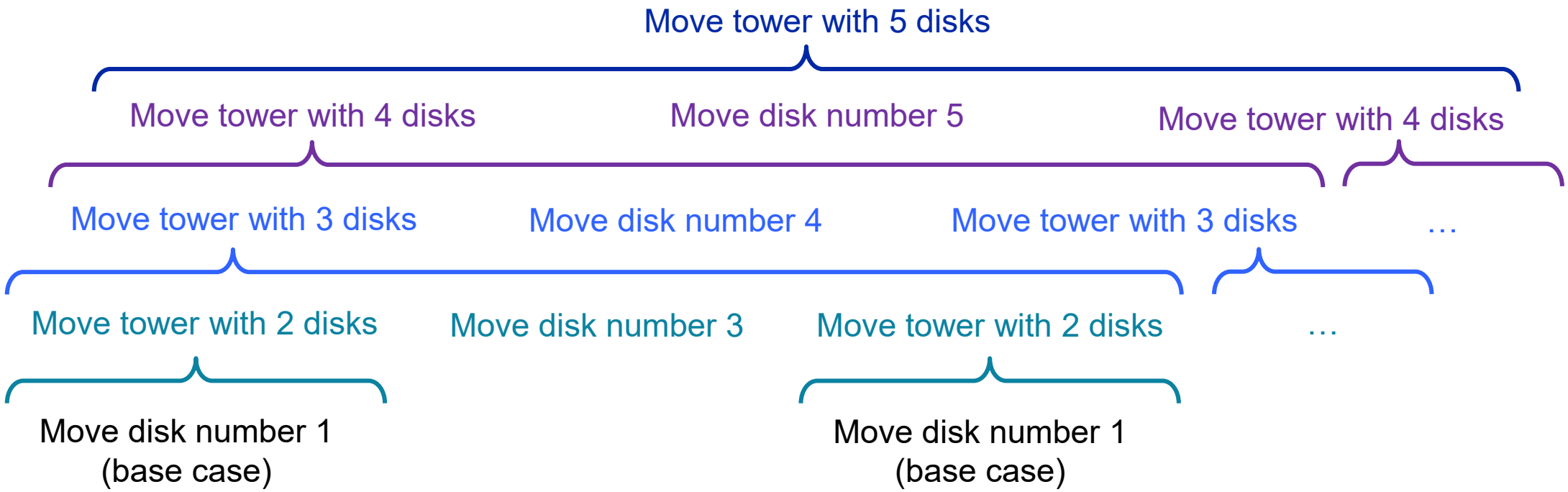
From this, we have a generalized, recursive solution to move a tower with k disks (i.e. disks from 1 to k) from place A to place C:

1) Move the tower with disks 1 to k − 1 from place A to place B
2) Move disk k (biggest disk of the tower of size k) on place C (only remaining valid place)
3) Move the tower with disks 1 to k − 1 from place B to place C

Steps 1) and 3) are applied recursively. The base case is reached when the tasks has reduced to shifting a tower of size 1, i.e. shifting a single disk.

Towers of Hanoi: Example with Five Disks

Graphical depiction of the recursive calls (recursion tree) for the example of five disks:



Towers of Hanoi: C Code Implementation

Possible implementation of a program which prints a sequence of movements which will solve the Towers of Hanoi problem:

```c
void printHanoiOperations(int numberOfDisks, char fromPlace, char viaPlace, char toPlace) {
   if (numberOfDisks == 1) {
      printf("%c -> %c\n", fromPlace, toPlace);
   }
   else {
      printHanoiOperations(numberOfDisks - 1, fromPlace, toPlace, viaPlace);
      printHanoiOperations(1, fromPlace, viaPlace, toPlace);
      printHanoiOperations(numberOfDisks - 1, viaPlace, fromPlace, toPlace);
   }
}
```