# Data Structures and Algorithms

UBCO Master of Data Science – DATA 532

## Recursion

## Recursive Thinking

***Recursion*** is:

- A *problem-solving* ***approach***, that can ...
- Generate *simple solutions* to ...
- *Certain kinds* of problems that ...
- Would be *difficult to solve in other ways*

Recursion *splits a problem*:

- Into one or more *simpler versions of* ***itself***

## Recursive Thinking

Consider the following list of numbers:

- `24, 88, 40, 37`

Such a list can be defined as follows:

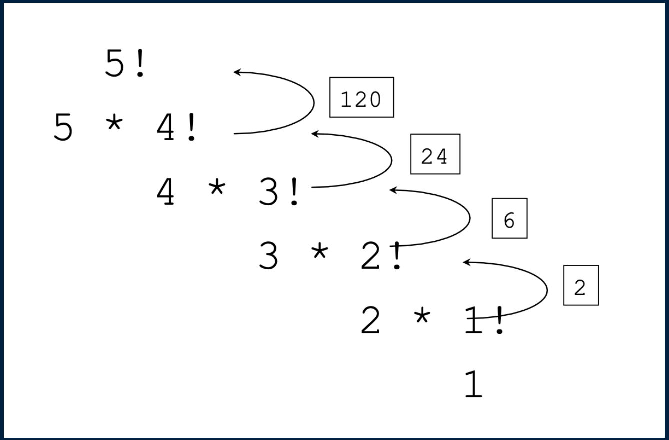
```
• A List is a: number
           or a: number comma List
```

The concept of a List is used to define itself

## Recursive Definitions

In mathematics, recursion is frequently used. The most common example is the factorial:

For example, 5! = 5(4)(3)(2)(1), or 5! = 5(4!)



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

## Recursion is a method in which the solution of a problem depends on _______________ ?

A) Larger instances of different problems

B) Larger instances of the same problem

C) Smaller instances of the same problem

D) Smaller instances of different problems

## Recursive Definition of factorial

In other words,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

This definition says that 0! is 1, while the factorial of any other number is that number times the factorial of one less than that number.

## Recursive Definitions

Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1. This is called a *base case* for the recursion.

When the base case is encountered, we get a closed expression that can be directly computed.

## Recursive Definitions

All good recursive definitions have these two key characteristics:

- There at least one base cases (or more) for which no recursion is applied.
- All chains of recursion eventually end up at one of the base case(s).

The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.

## Recursive Functions

If factorial is written as a separate function:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

## Example: String Reversal

Python lists have a built-in method that can be used to reverse the list. What if you wanted to reverse a string?

If you wanted to program this yourself, one way to do it would be to convert the string into a list of characters, reverse the list, and then convert it back into a string.

```
def split(word):
   x=list(word)
   x.reverse();
   return x
print(split("Hello"))
```

```
['o', 'l', 'l', 'e', 'H']
```

## Example: String Reversal

Using recursion, we can calculate the reverse of a string without the intermediate list step.

Think of a string as a recursive object:

- Divide it up into a first character and “all the rest”
- Reverse the “rest” and append the first character to the end of it

## Example: String Reversal

```
def reverse(s):
    return reverse(s[1:]) + s[0]
```

The slice `s[1:]` returns all but the first character of the string.

We reverse this slice and then concatenate the first character (`s[0]`) onto the end.

## Example: String Reversal

```
>>> reverse("Hello")

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    reverse("Hello")
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
…
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
RuntimeError: maximum recursion depth exceeded
```

What happened? There were 1000 lines of errors!

## Example: String Reversal

Remember: To build a correct recursive function, we need a base case that doesn’t use recursion.

We forgot to include a base case, so our program is an *infinite recursion*. Each call to `reverse` contains another call to `reverse`, so none of them return.

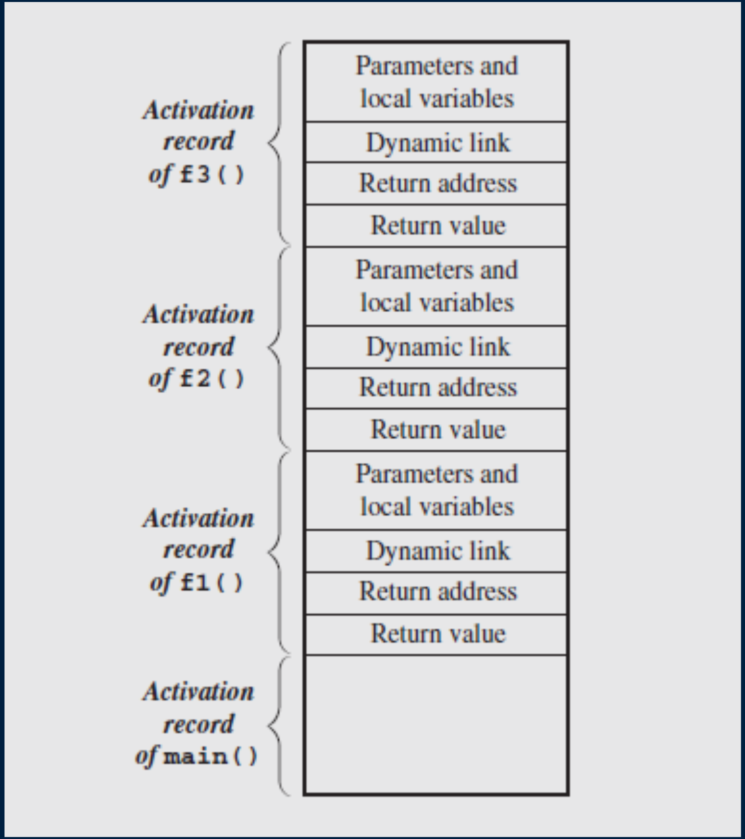
## Example: String Reversal

Each time a function is called it takes some memory. Python stops it at 1000 calls, the default "maximum recursion depth."

What should we use for our base case?

Following our algorithm, we know we will eventually try to reverse the empty string. Since the empty string is its own reverse, we can use it as the base case.

## Contents of the run-time stack when main program (main() ) calls function f1(), f1() calls f2(), and f2() calls f3()


Activation record of f3()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f2()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f1()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of main()

## Example: String Reversal

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]
print(reverse("Hello"))

The output is:
olleH
```

## Example: Binary Search

Now that you’ve seen some recursion examples, you’re ready to look at doing binary searches recursively.

Remember: we look at the middle value first, then we either search the lower half or upper half of the array.

The base cases are when we can stop searching, namely, when the target is found or when we’ve run out of places to look.

## What will happen if the  base case is missing in recursive functions?

A) The code will be executed successfully and no output will be generated

B) The code will be executed successfully and random output will be generated

C) The code will show a compile time error

D) The code will run for some time and stop when the stack overflows

## Hard Problems

Using divide-and-conquer we could design efficient algorithms for searching and sorting problems.

Divide and conquer and recursion are very powerful techniques for algorithm design.

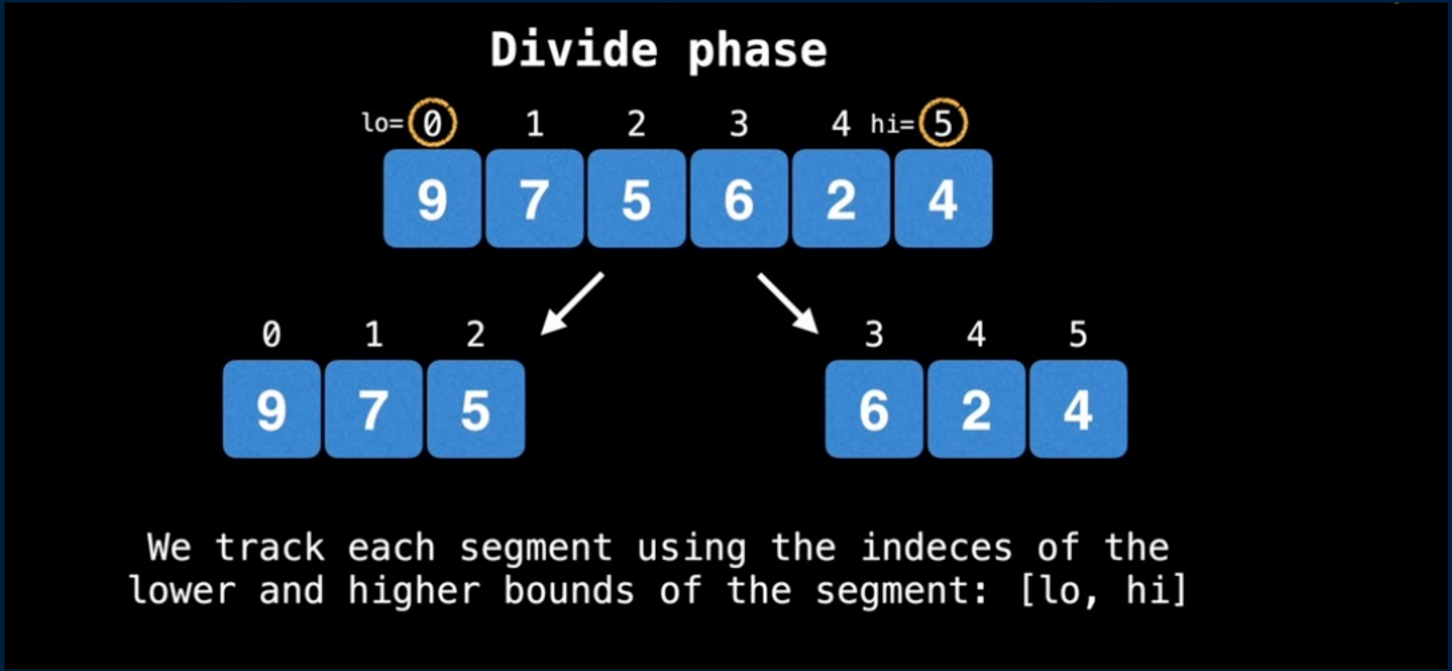
Not all problems have efficient solutions!

## MERGE SORT Algorithm

*Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted).*

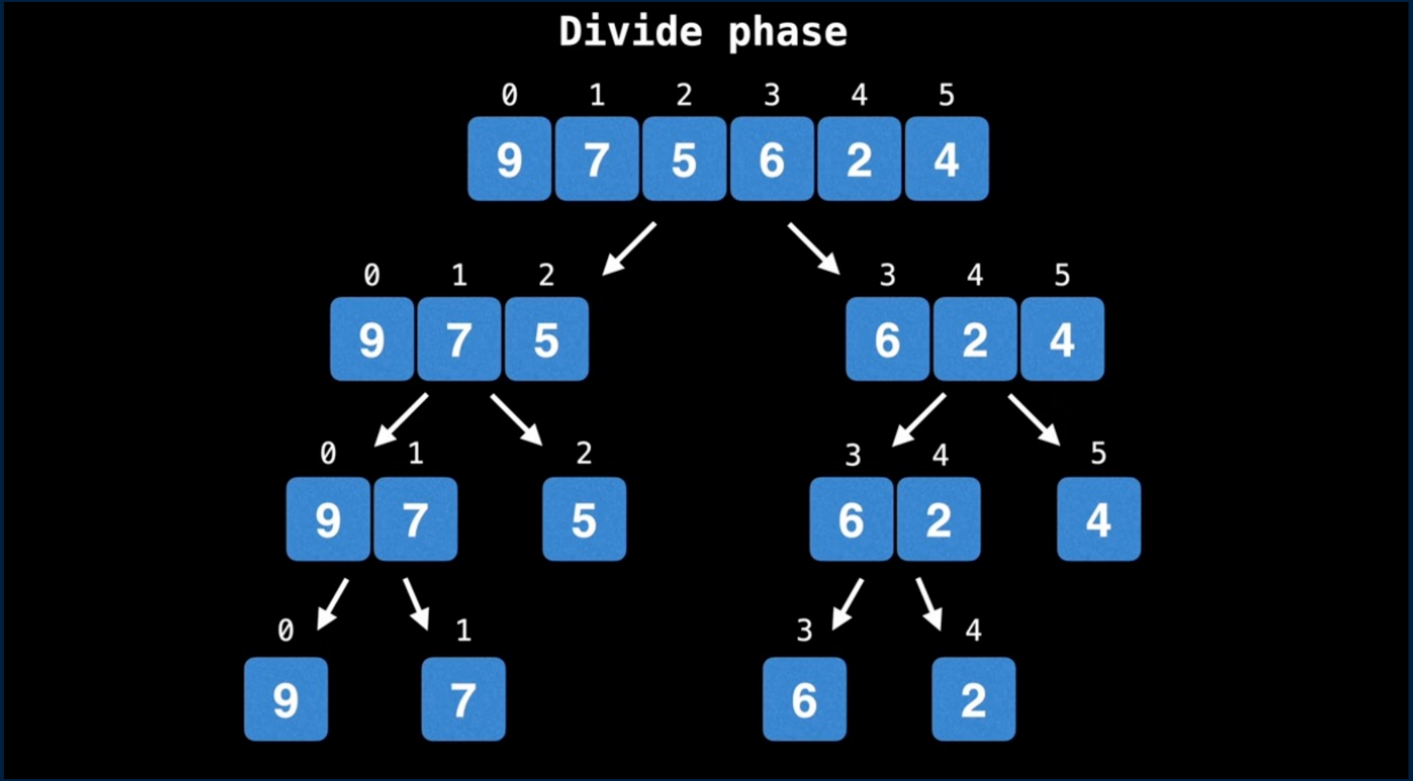
*Then the sorted subarrays are merged into one sorted array.*

## MERGE SORT  Algorithm

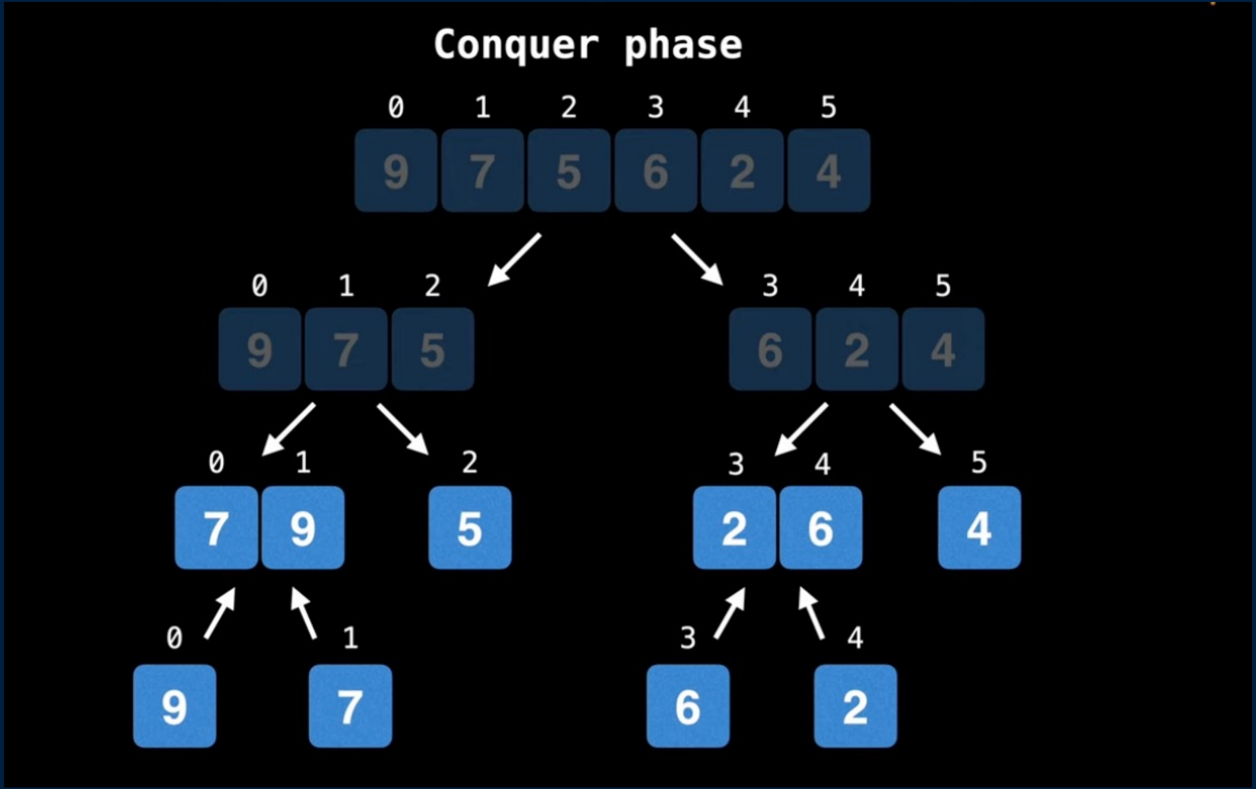


We track each segment using the indeces of the lower and higher bounds of the segment: [lo, hi]

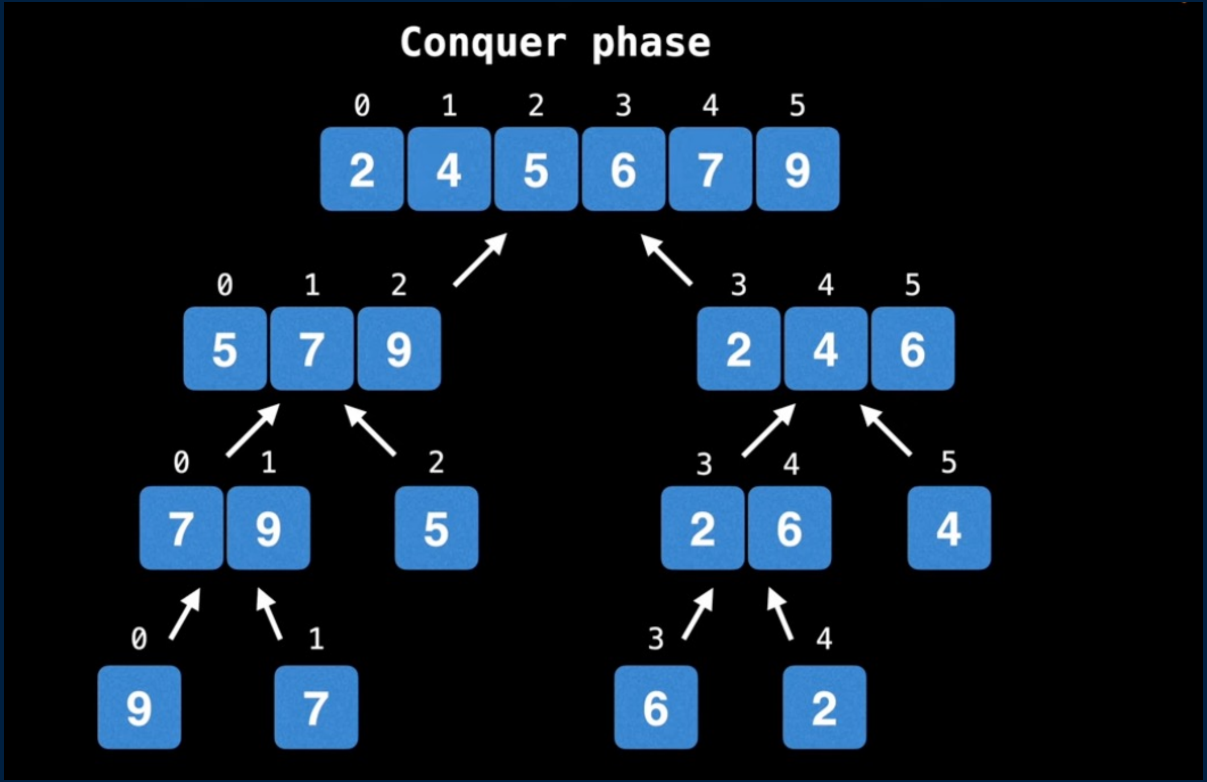
## MERGE SORT  Algorithm



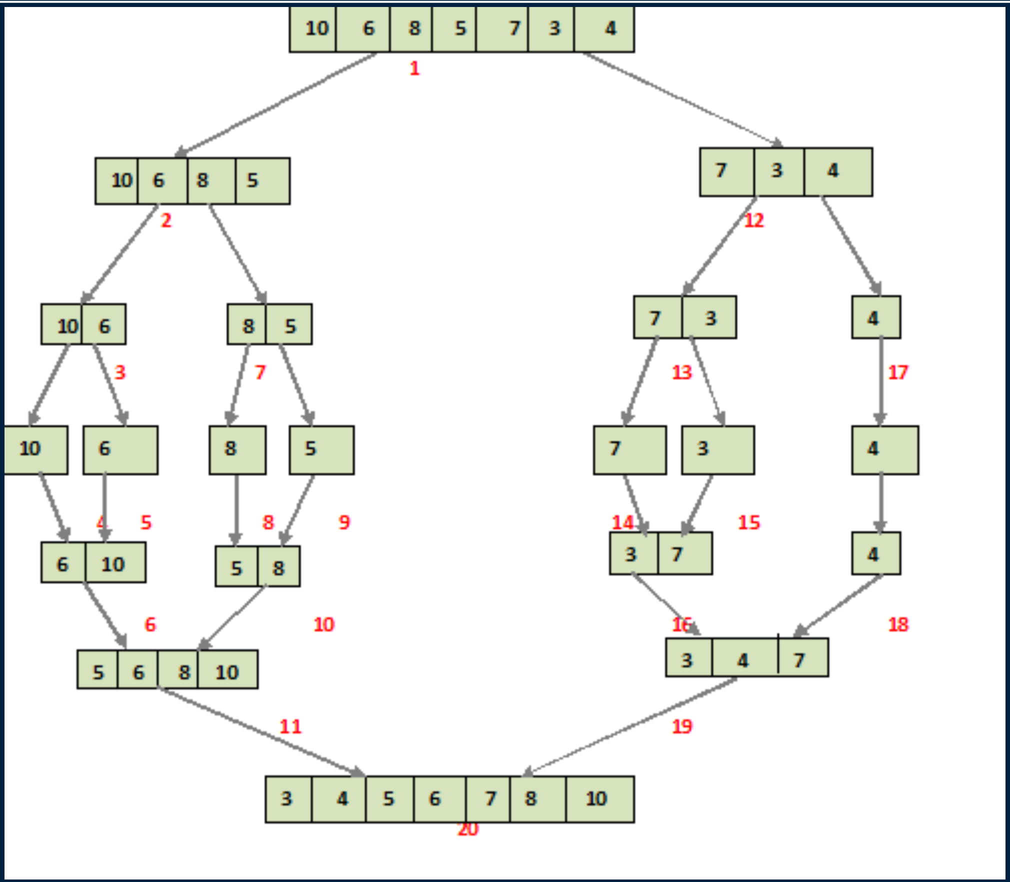
# MERGE SORT  Algorithm



# MERGE SORT  Algorithm



# MERGE SORT  Algorithm



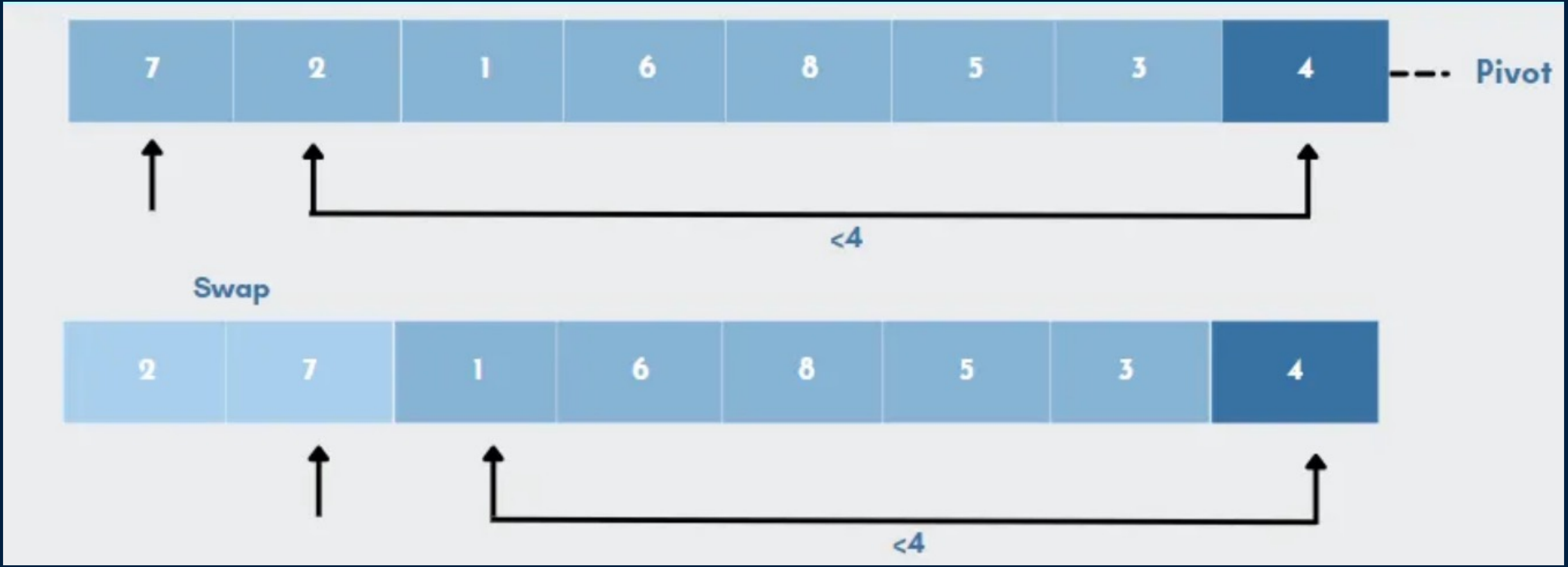
## QUICK SORT  Algorithm

Quicksort is a fast sorting algorithm that works by splitting a large array of data into smaller sub-arrays. **This implies that each iteration works by splitting the input into two components, sorting them, and then recombining them.**

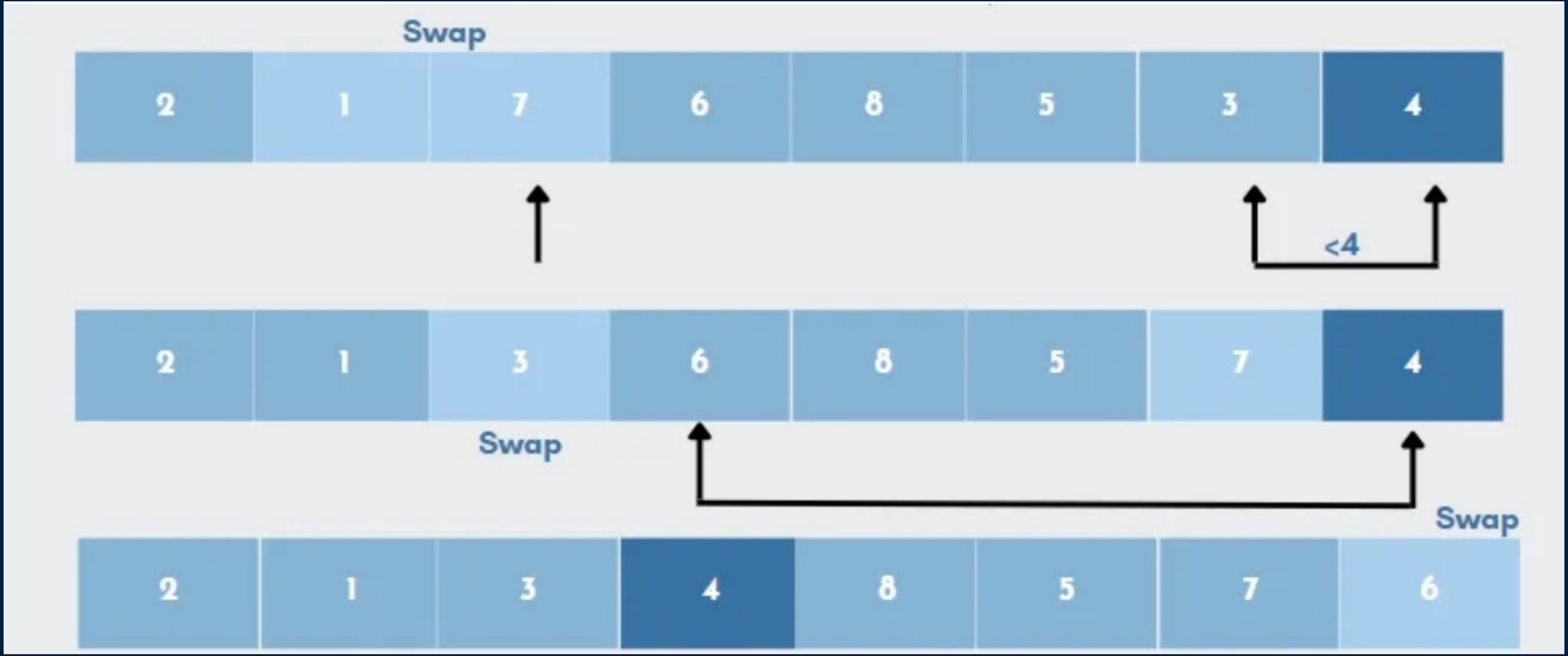
## QUICK SORT  Algorithm

- **Pivot:** Select an element.
- **Divide:** Split the problem set, move smaller parts to the left of the pivot and larger items to the right.
- **Repeat and combine:** Repeat the steps and combine the arrays that have previously been sorted.

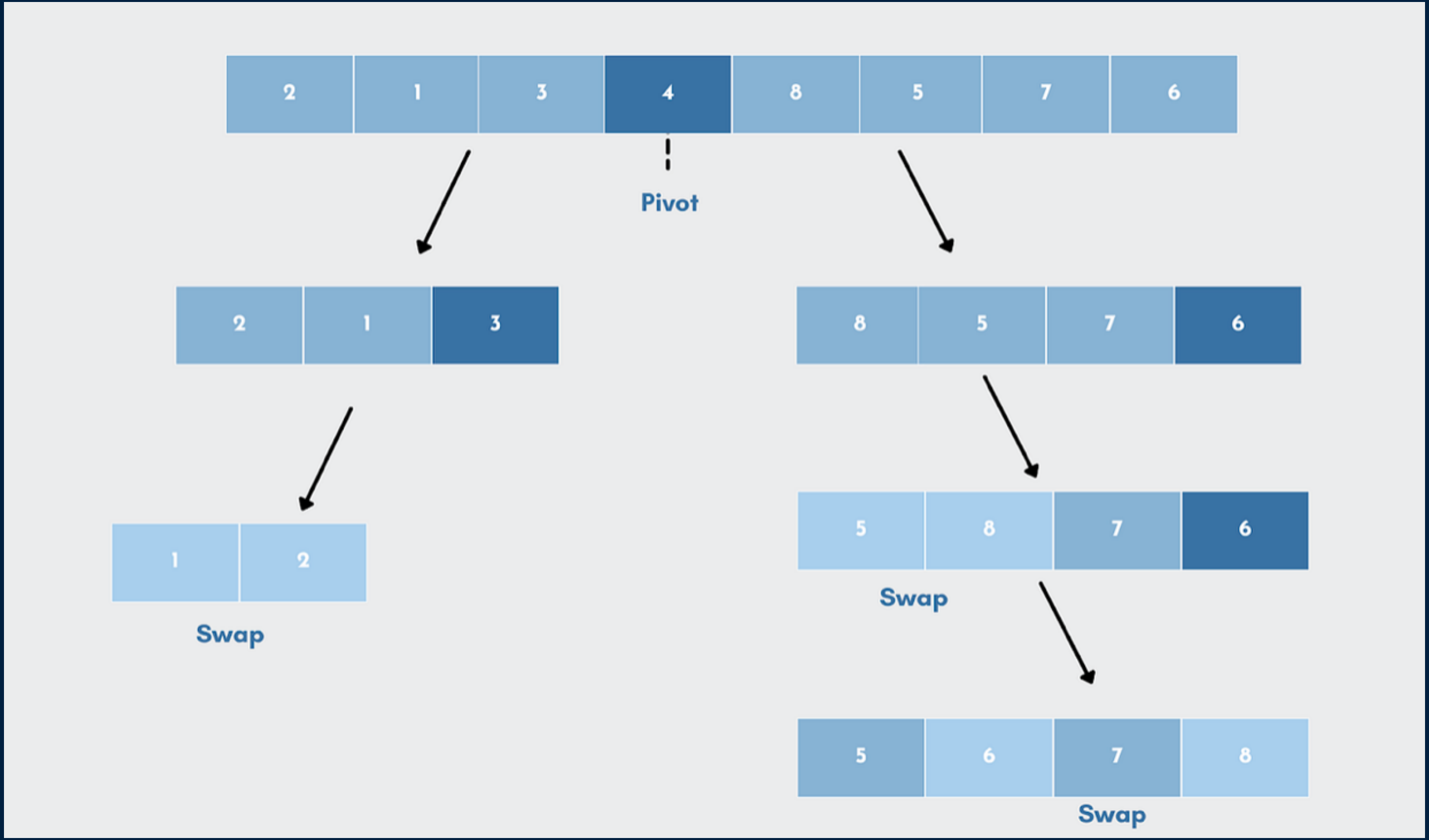
# QUICK SORT Algorithm



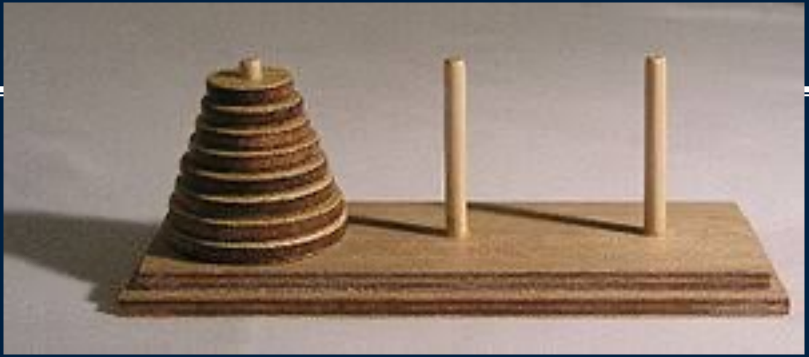
# QUICK SORT Algorithm



## QUICK SORT  Algorithm



# The Towers of Hanoi



The Legend:

> *"At the beginning of time, an ancient brotherhood of monks was established and given a sacred task. They were charged with moving 64 golden disks from one peg to another. In addition to the source and destination pegs is a third peg which can be used to temporarily place disks. The disks are stacked with the largest on bottom and smallest on top. No disk can be placed on top of a smaller disk, else the disks will break. It is said that when the monks complete their task, the world will crumble into dust and end…"*

*A*ssuming that the monks are happy about this, how should they go about this task?

There are 3 pegs (A, B, and C) and n disks of different diameters.

Starting position: All disks on peg A, with each disk resting on a larger disk (except for the bottom disk, which is the largest one).

The goal: Move all disks from peg A to peg B, sorted in the same manner from bottom to top.

Rules:

- Only one disk can be moved at each turn.
- Only the top disk on a peg can be moved.
- A disk may only be moved to another peg.
- A larger disk can never be placed on a smaller disk.

If you have one disk on peg A, move it to peg B. Otherwise...

- Move the top n-1 disks on peg A to peg C
- Move the one disk left on peg A to peg B
- Move the top n-1 disks on peg C to peg B.

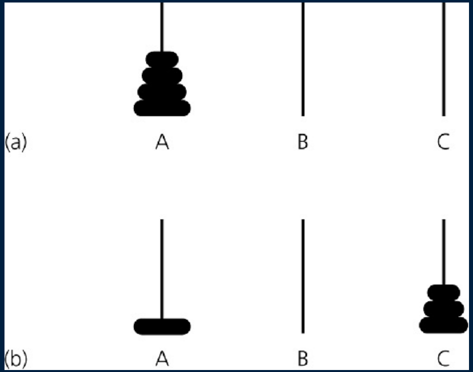
- How can these underlined steps be carried out? Recursion to the rescue!

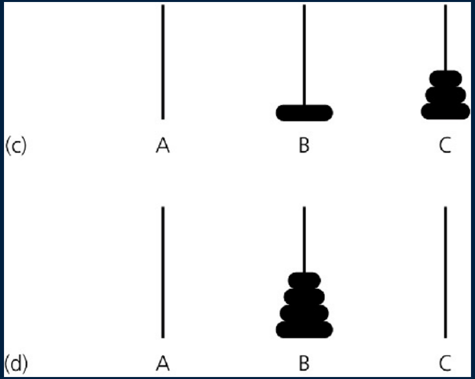
(a) Starting position.

(b) Move the top 3 disks on peg A to peg C.

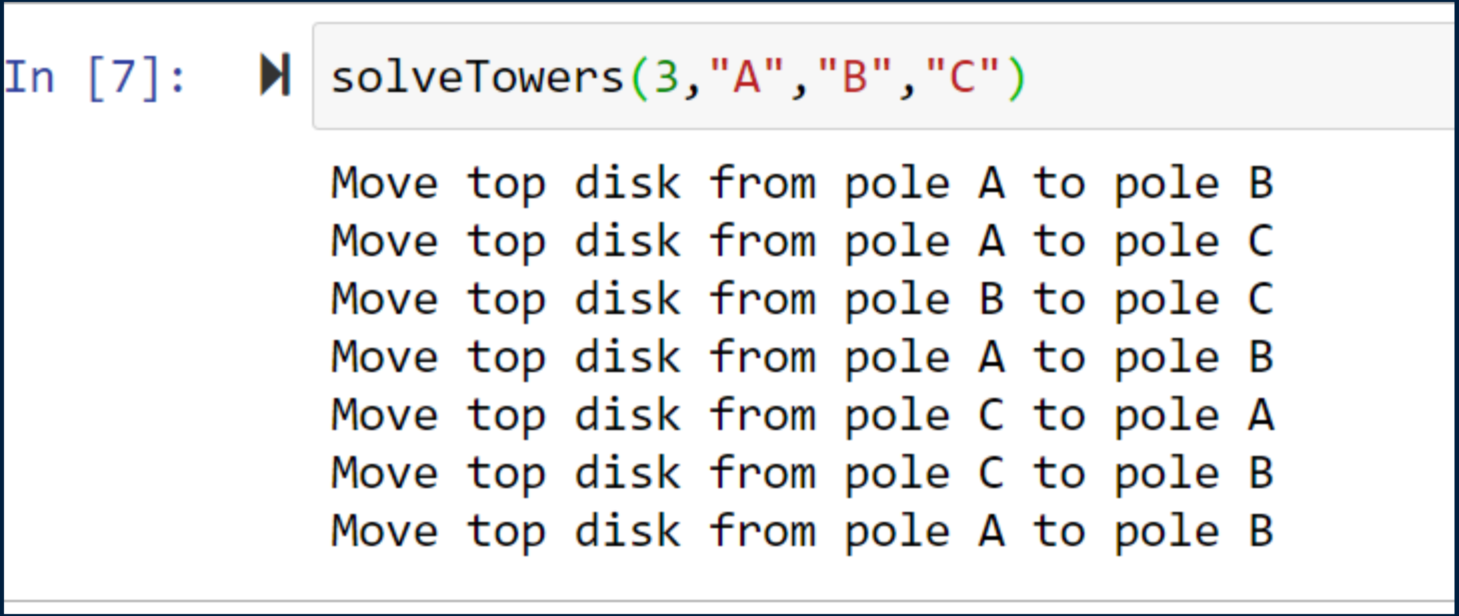
(c) Move the one disk left on peg A to peg B.

(d) Move the top 3 disks on peg C to peg B.





## Sample execution



```
In [7]:   solveTowers(3,"A","B","C")

          Move top disk from pole A to pole B
          Move top disk from pole A to pole C
          Move top disk from pole B to pole C
          Move top disk from pole A to pole B
          Move top disk from pole C to pole A
          Move top disk from pole C to pole B
          Move top disk from pole A to pole B
```

## Question

How much time will it take to move the 64 golden disks?

A) Order of minutes

B) Order of hours

C) Order of months

D) Order of years

E) Order of millions of years

F) I do not have a clue

## Complexity analysis

Let’s analyze this recursive solution using the same pseudocode

Let moves(n) be the number of required moves by the function

- If n=1 then moves(1) = 1
- If n>1 then moves(n) is given by

  moves(n)=moves(n-1) + moves(1) + moves (n-1)

  = 2 moves(n-1) +1

Hence:

- moves(1) = 1
- moves(2) = 2 moves(1) + 1 = 3
- moves(3) = 2 moves(2) + 1 = 7
- Moves(4) = 2 moves(3) +1 = 15
- And so on

This pattern is equivalent to:

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

**Pseudo Code:**

```
solveTowers(count , source , destination , spare) :
  if (count is 1 ) :
        Move a disk directly from source to destination
  else:
        solveTowers(count-1, source , spare , destination)
        solveTowers(1 , source , destination , spare)
        solveTowers(count-1, spare , destination , source)
```

Assume the monks are really fast: They move one disk every second.
How many seconds will it take to move the 20 golden disks?

moves(20) = $2^{20}$ - 1 seconds
= 1048576 seconds
= 1048576/(60 * 60 * 24 ) days
= 12 days

How many seconds will it take to move the 64 golden disks?

moves(64) = $2^{64}$ - 1 seconds
= 18446744073709551615 seconds
= 18446744073709551615/(60 * 60 * 24 * 365) years
= 584942417355 years
= 584 billion years

## Recap...

- Notion of recursive functions and algorithms
- Different applications of recursion
- Merge sort and quick sort algorithms
- Compared a recursive and iterative algorithm

THE UNIVERSITY OF BRITISH COLUMBIA