Problem Solving Strategies and Computational Approaches SCS1304

Handout 2 : Introduction to Computational Thinking

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

Lecture Overview

- Introduction to Problem Solving Strategy
  - Linear Search
  - Binary Search
  - Fibonacci Sequence
  - Introduction to Recursion
  - Types of Recursion
  - Applications of Recursion
  - Best Practices and Pitfalls in using Recursion in Algorithmic Design
  - Recursion vs Loops/Iteration
- Analysis of Algorithms

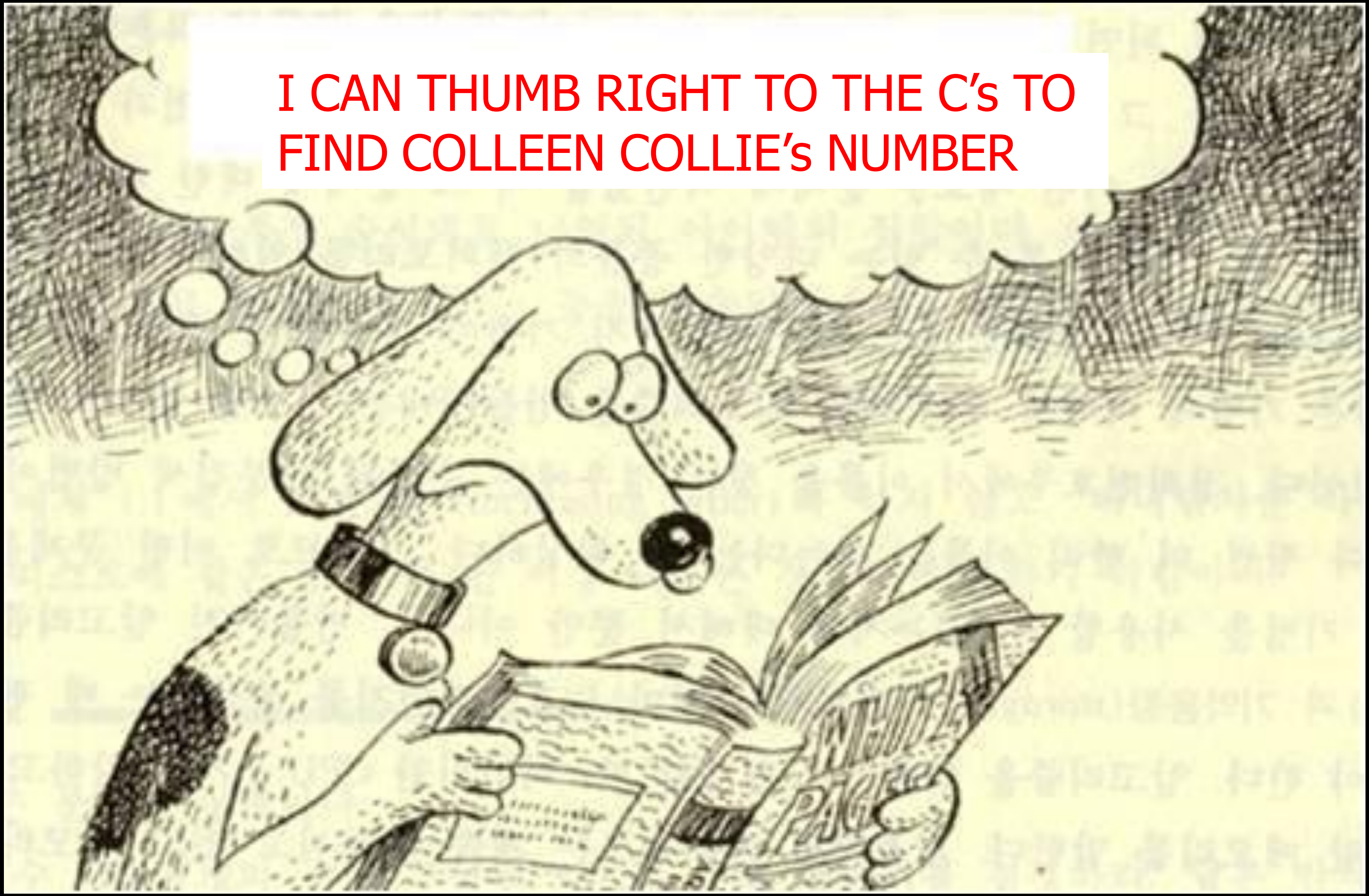
Algorithms : Efficiency, Analysis, And Order [Ref Chapter 1 of recommended book]

Goals of up coming lectures ..

- Design
  - Problem solving method
  - 5 design methods: divide-and-conquer, dynamic programming, greedy approach, backtracking, branch-and-bound
- Analysis
  - Efficiency analysis through computational complexity

Acknowledgements

-
    - Notes adopted from Neapolitan, R. and Naimipour, K., 2015. Foundations of Algorithms.
    - Other Reference acknowledged in slides where appropriate


I CAN THUMB RIGHT TO THE C's TO FIND COLLEEN COLLIE's NUMBER
WHITE PAGES

Example

- Problem
  - Determine whether the number x is in the list of S of n numbers. The answer is yes if x is in S and no if it is not.
  - Problem instance
    - S = [10, 7, 11, 5, 13, 8], n = 6, x = 8
    - Solution to this instance is, “Yes, x is in S”
- Algorithm
  - Starting with the first item in S, compare x with each item in S in sequence until x is found or S is exhausted. If x is found, answer yes; if x is not found, answer no.
- Analysis
  - Any *better* algorithm to get the same solution?

Problem Description

- Problem
  - May contain variables that are assigned specific values in the statement of the problem description.
- Parameters
  - Those variables are called parameters.
  - For example, to describe a search problem, we need 3 variables: S, n, x.
- Instance
  - If those parameters are specified, we call it an instance.
  - S = [10, 7, 11, 5, 13, 8], n = 6, x = 8
- Solution to an instance of a problem is the answer to the question asked by the problem in that instance.
  - Solution to the above instance is, “yes, x is in S”

Algorithmic Description

- Natural languages
- Programming languages
- Pseudo-code
    - similar, but not identical, to C/C++/Java.
    - Notable exceptions: unlimited array index, variable-sized array, mathematical expressions, use of arbitrary types, convenient control structure, and etc.
- In this course unit, algorithms will be represented by pseudo-codes similar to C/C++ & flow charts.

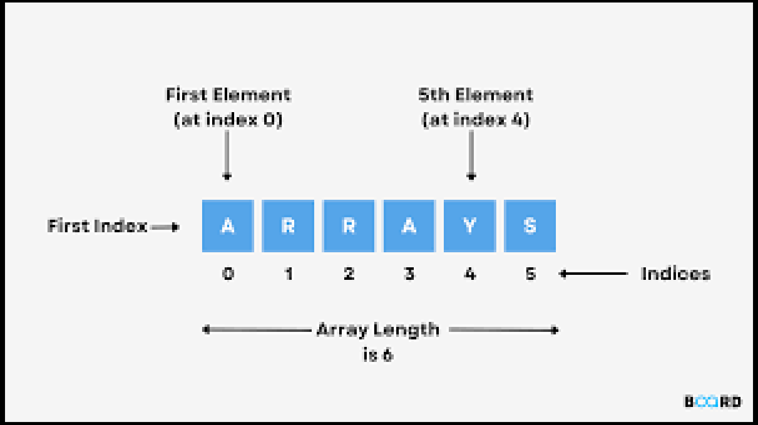
Pseudo-code vs. C/C++

- Use of arrays
  - In C/C++, starting at 0
  - In pseudo-code, arrays indexed by other integers are ok.
- Variable-sized array size
  - (e.g) void example (int n)
    - { keytype S[2..n];
      - ….
    - }
  - keytype S[low..high]

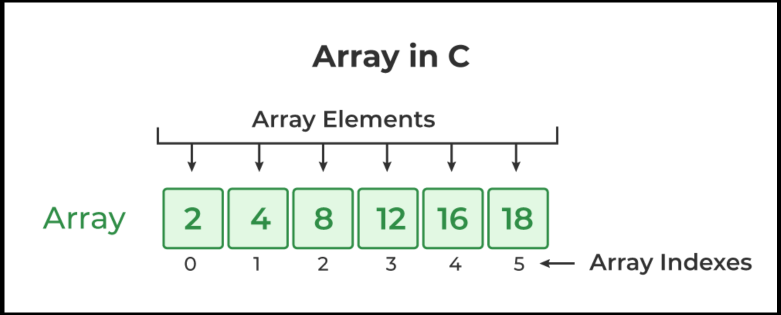
Pseudo-code vs. C/C++ ctd..

- Mathematical expression
  - `low <= x && x <= high` ⇒ low ≤x ≤ high
  - `temp = x; x = y; y = temp` ⇒ exchange x and y
- Use of arbitrary type
  - Index
  - Number
  - Bool
- Control structure
  - (e.g.) `repeat (n times) { … }`

Sequential Search



- Problem
  - Is the key 'x' in the array S of n keys?
- Inputs (parameters)
  - Positive integer n, array of keys S <u>indexed from 1 to n</u>.
- Outputs
  - The location of x in S. (0 if x is not in S.)

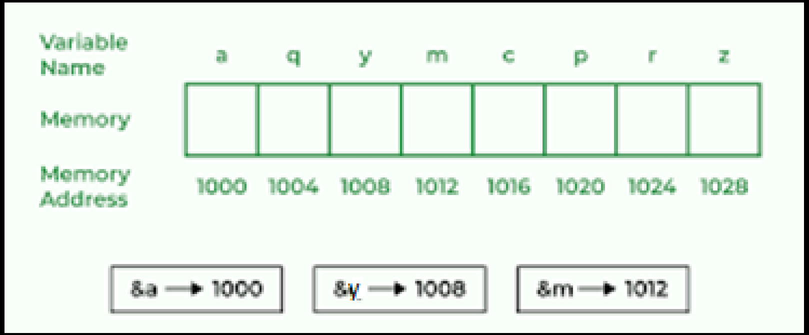


- Algorithm
  - Starting with the first item in S, compare x with each item in S in sequence until x is found or S is exhausted. If x is found, answer yes; if x is not found, answer no.

Sequential Search (Psedo-code)

```
void seqsearch(int  n,              // Input(1)
               const keytype S[],       //  (2)
               keytype x,               //  (3)
               Index &location){        // Output
 location = 1;
  while (location <= n && S[location] != x){
     location++;
  }
  if (location > n)
     location = 0;
}
```

'const' keyword in C is used to define constant values that cannot be changed at runtime



Memory Address &

When a variable is created in C, a memory address is assigned to the variable.
The memory address is the location of where the variable is stored on the computer.
When we assign a value to the variable, it is stored in this memory address.
To access it, use the reference operator (&), and the result represents where the variable is stored:

https://www.w3schools.com/c/c_memory_address.php

Review: sequential search

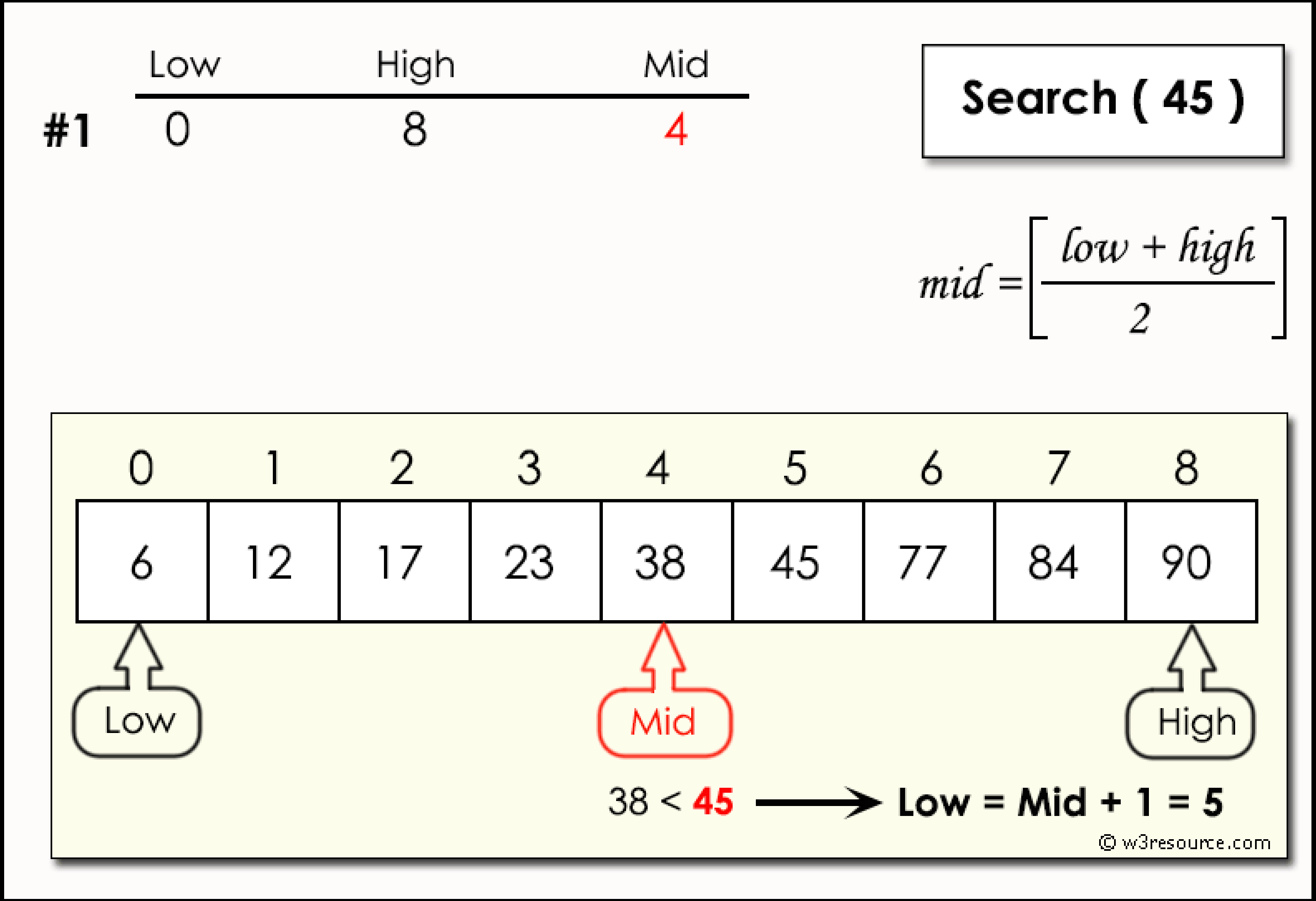
- How many comparisons for searching x in S?
  - Depends on the location of x in S
  - In worst case, we should compare n times.
  - In best case, 1

- Any *better* algorithm to get the same solution?
  - No, we can’t, unless there is an extra information in S.

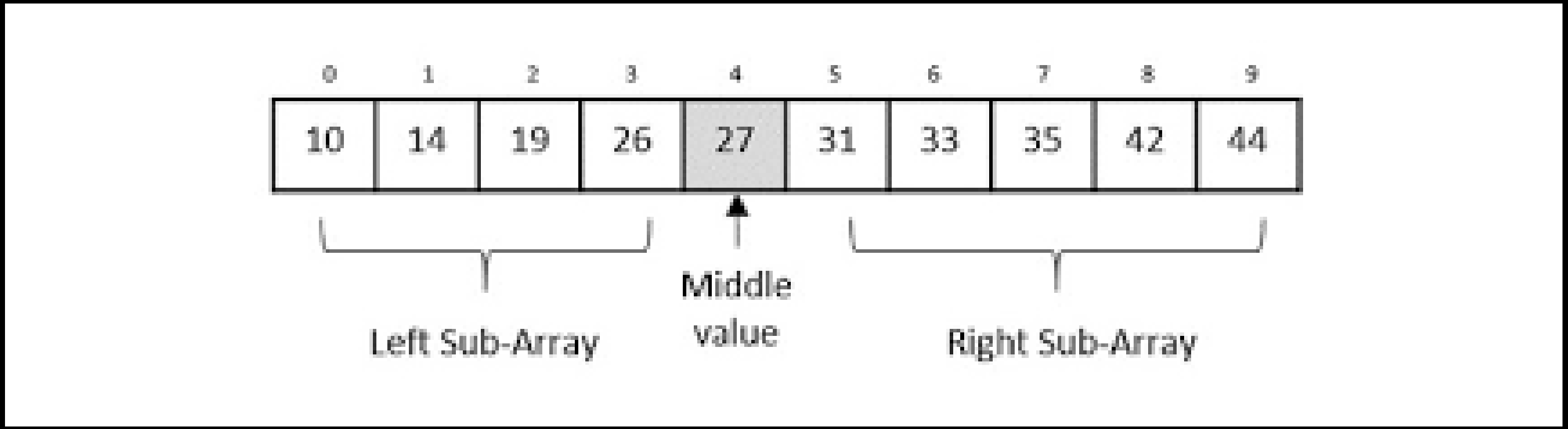
Binary Search

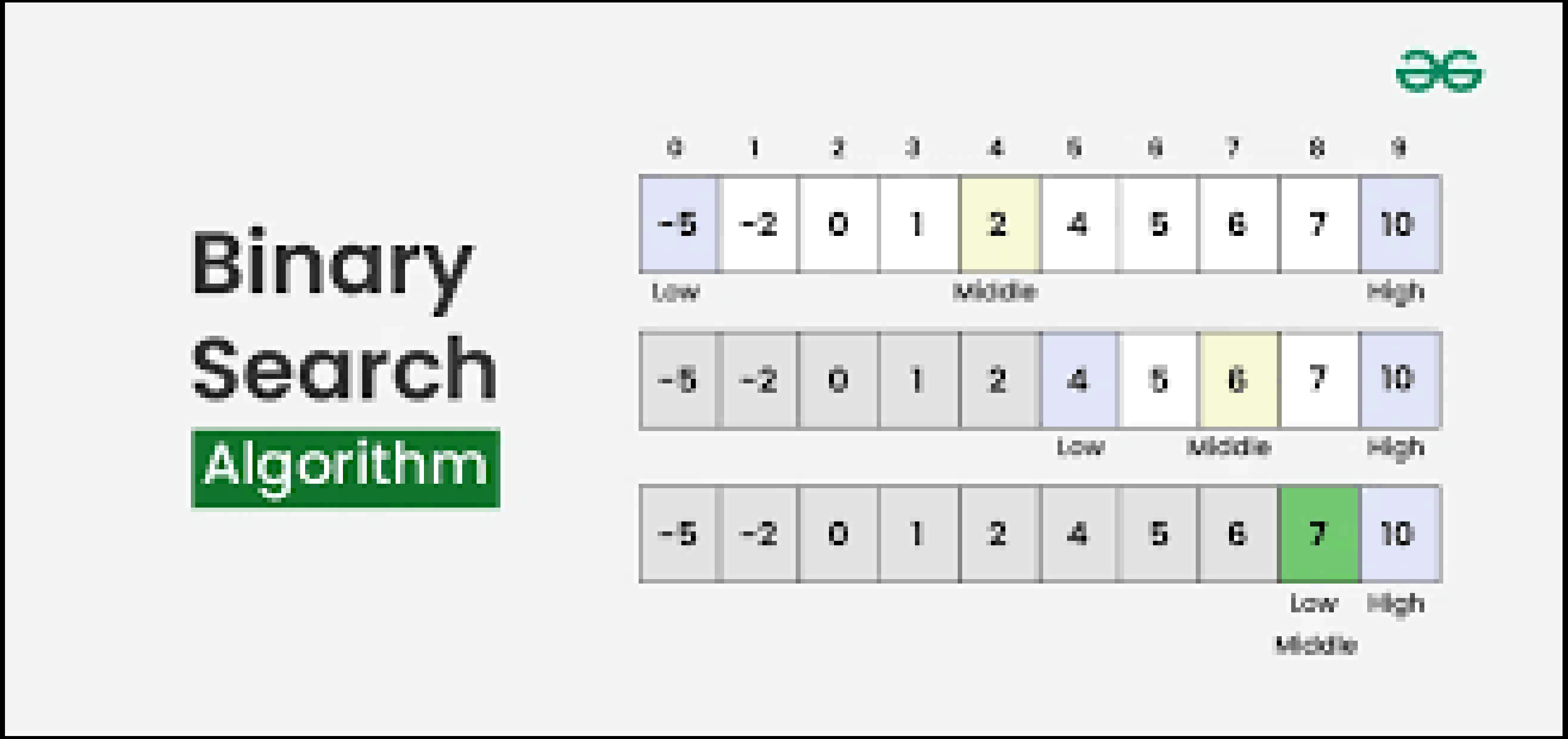
- Problem
  - Is the key x in the array S of n keys?
  - Determine whether x is in the sorted array S of n keys.
- Inputs (parameters)
  - Positive integer n, sorted (non-decreasing order) array of keys S indexed from 1 to n, a key x.
- Outputs
  - The location of x in S. (0 if x is not in S.)

Binary Search



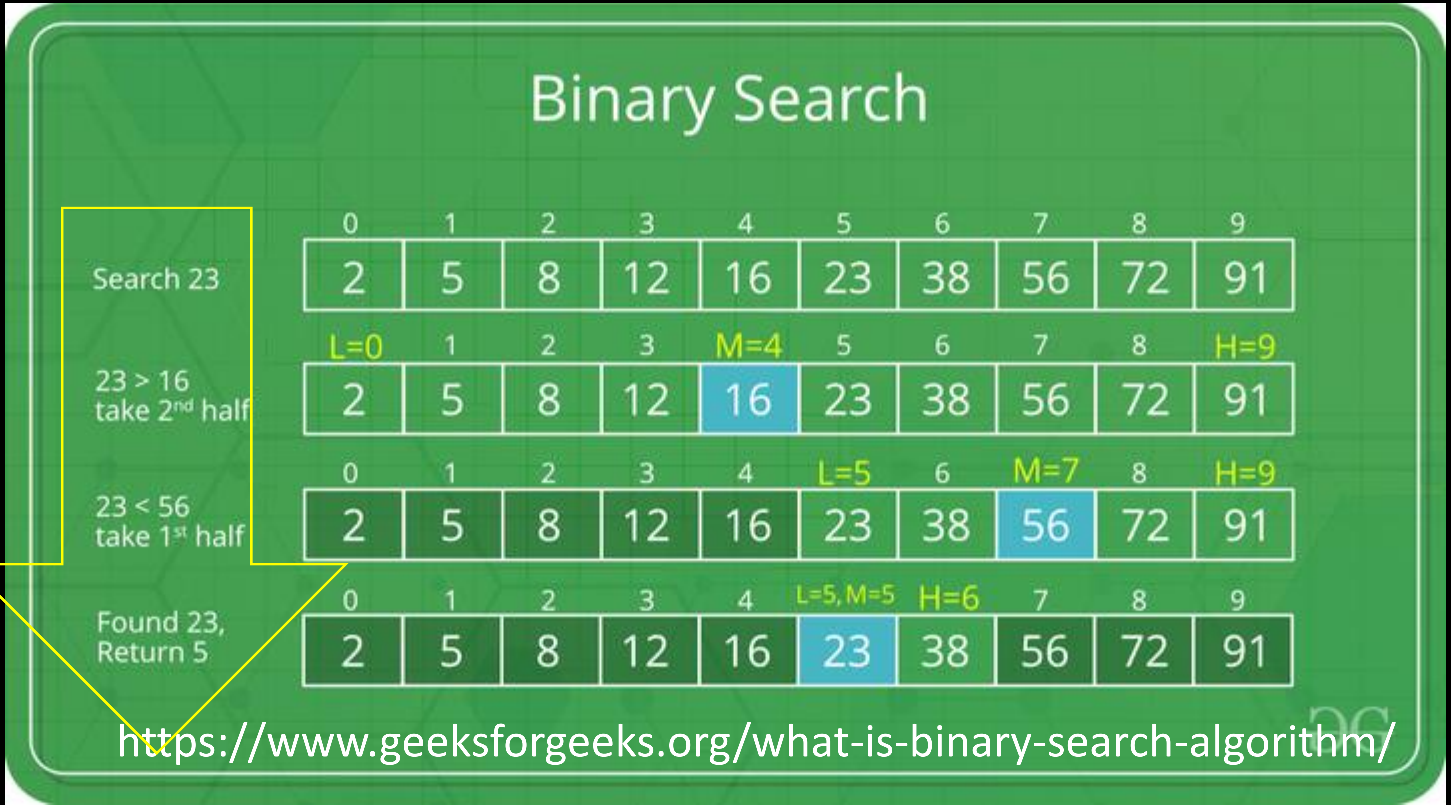
Binary Search





Properties of Binary Search

- Binary search is performed on the sorted data structure for example sorted array.
- Searching is done by dividing the array into two halves
- It utilizes the divide-and-conquer (discussed later) approach to find an element.



	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91

https://www.geeksforgeeks.org/what-is-binary-search-algorithm/

Binary Search (Psedo-code)

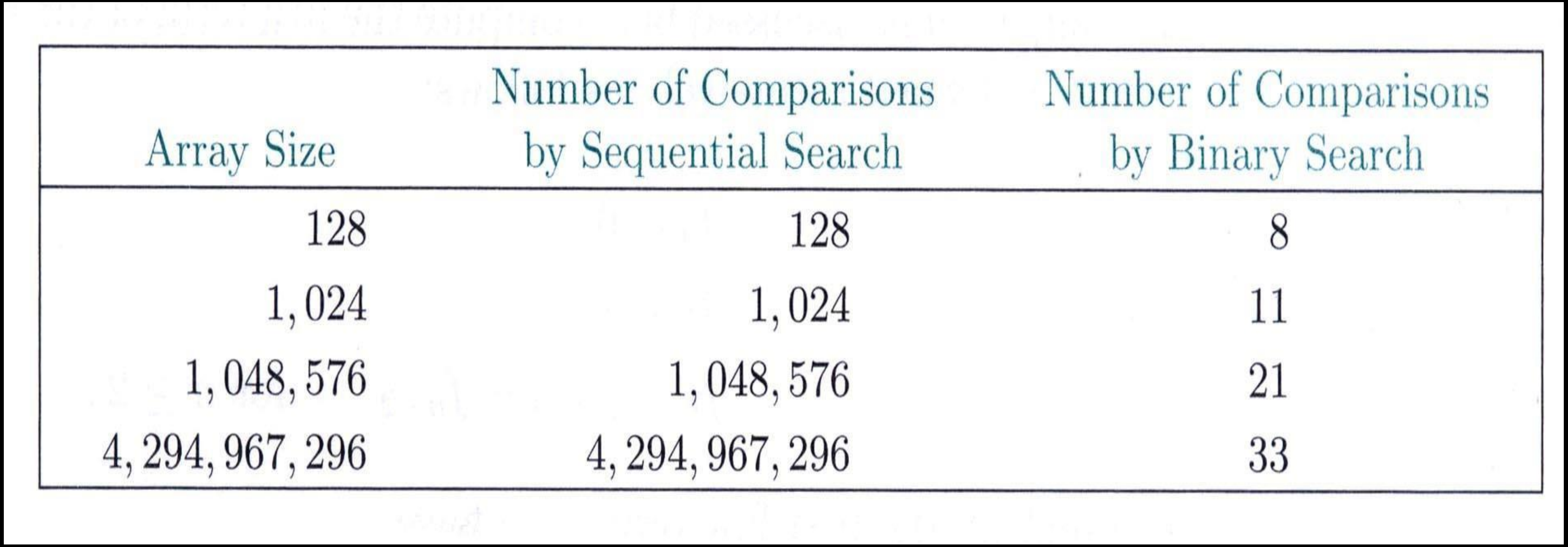
```
void binsearch(int n,                // Input(1)
               const keytype S[],   // (2)
               keytype x,           // (3)
               index& location) {   // Output
  index low, high, mid;
  low = 1; high = n;
  location = 0;
  while (low <= high && location == 0) {
      mid = (low + high) / 2;    // integer div
      if (x == S[mid])
          location = mid;
      else if
          (x < S[mid]) high = mid - 1;
      else
          low = mid + 1;
    }
  }
```

Refer: Bin Search Animation
https://yongdanielliang.github.io/animation/web/BinarySearchNew.html

Review: binary search algorithm

- How many comparisons for searching *x* in *S*?
  - *S* is already sorted. We know it.
  - For each statement in a while-loop, the number of searching targets will be half.
  - In worst case, we should compare (?) times.
    - (e.g.) n = 32. S[16], S[16+8], S[24+4], S[28+2], S[30+1]. S[32]
  - In best case, trivial to answer.
- Any *better* algorithm to get the same solution?
  - See Chap. 7-8.

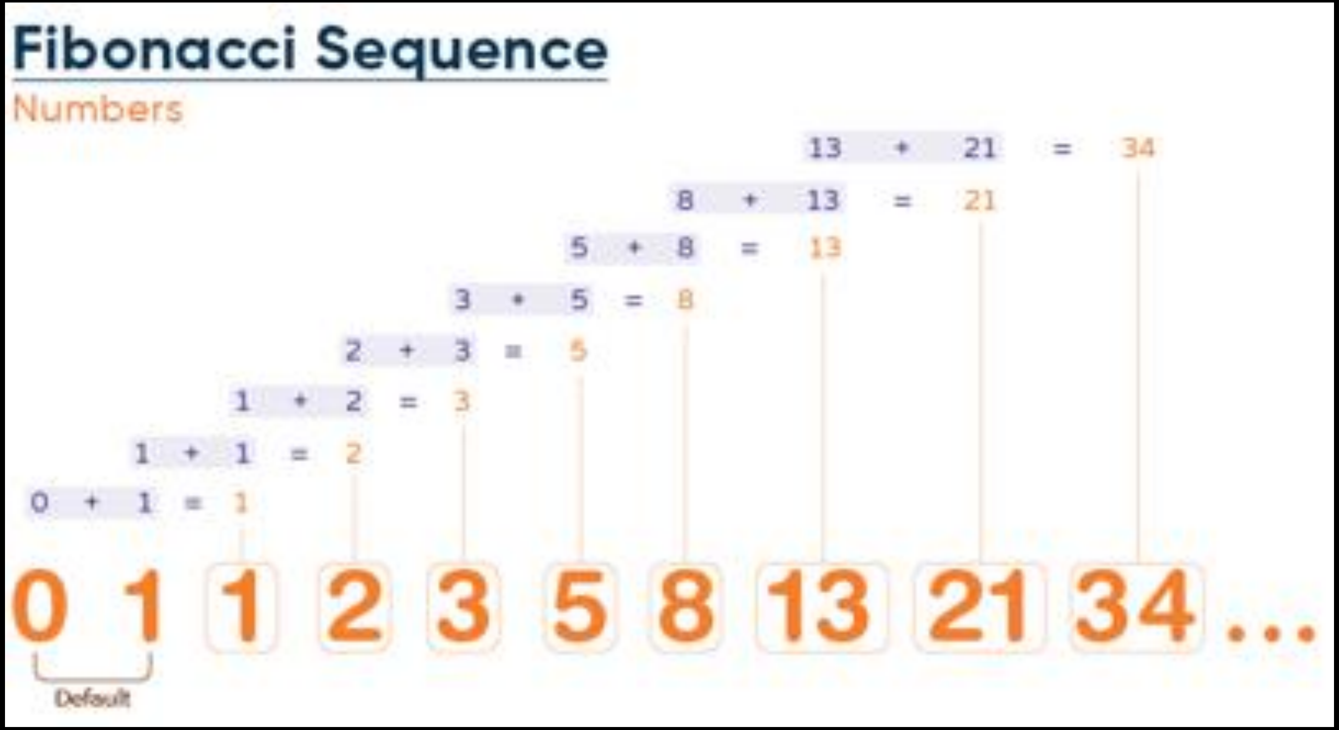
Table 1.1 The number of comparisons done by Sequential Search and Binary Search when x is larger than all the array items



Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

Fibonacci Sequence

- The Fibonacci Sequence is a number series in which each number is obtained by adding its two preceding numbers. It starts with 0 and is followed by 1.
- The numbers in this sequence, known as the Fibonacci numbers, are denoted by Fn.



https://mathmonks.com/fibonacci-sequence

Fibonacci Sequence

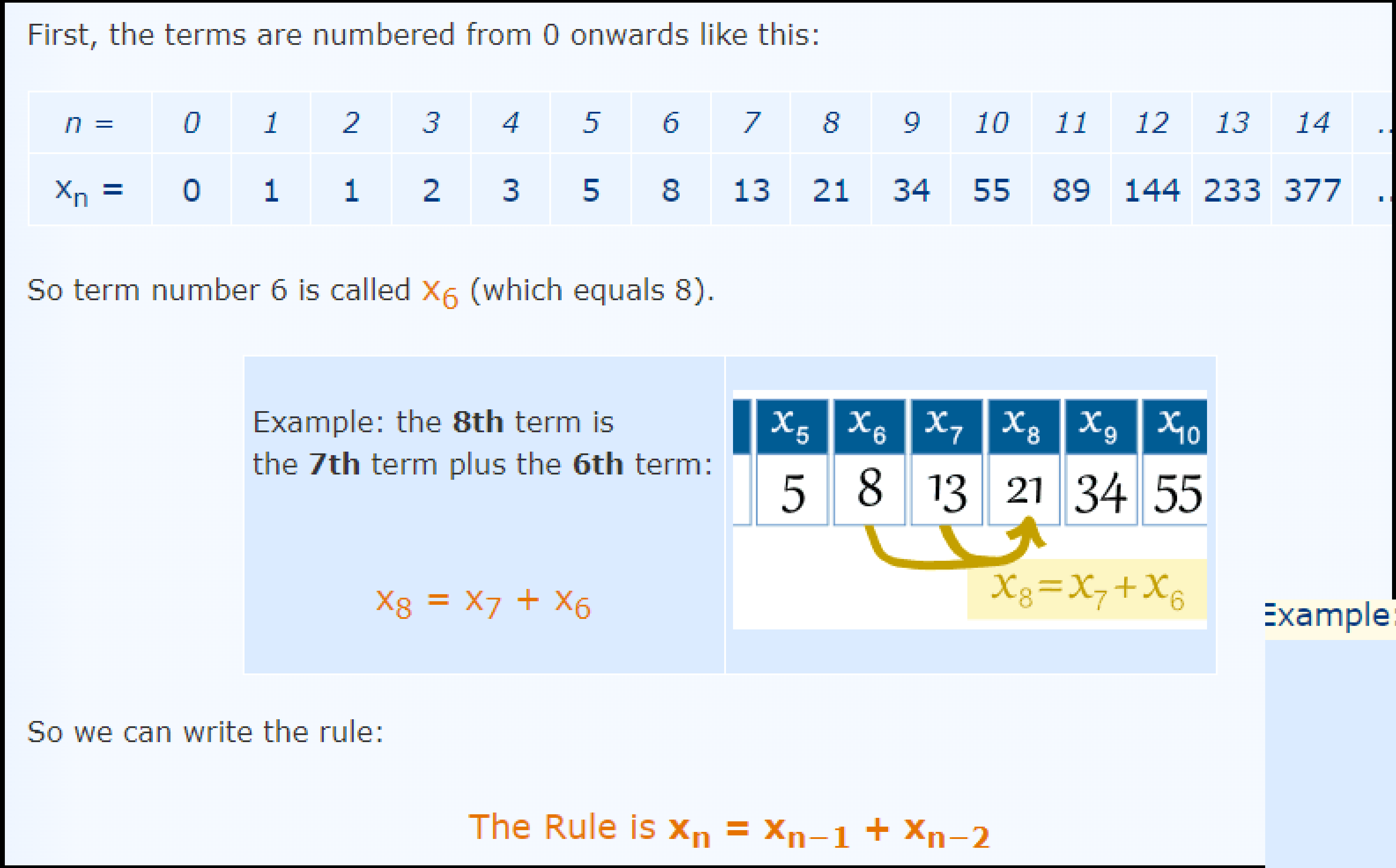
- The Fibonacci Sequence is the series of numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- The next number is found by adding up the two numbers before it:
  - the 2 is found by adding the two numbers before it (1+1),
  - the 3 is found by adding the two numbers before it (1+2),
  - the 5 is (2+3),
  - and so on!
- Example: the next number in the sequence above is 21+34 = 55

https://www.mathsisfun.com/numbers/fibonacci-sequence.html

Fibonacci Sequence



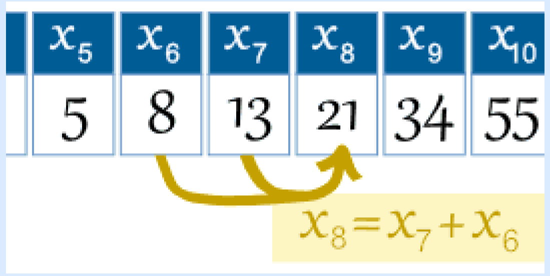
First, the terms are numbered from 0 onwards like this:

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..
$x_n =$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	..

So term number 6 is called x_6 (which equals 8).

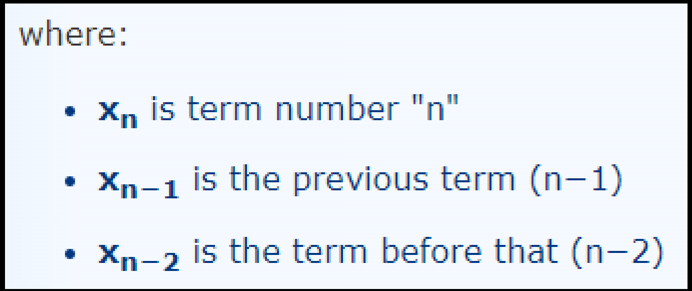
Example: the **8th** term is the **7th** term plus the **6th** term:

$x_8 = x_7 + x_6$



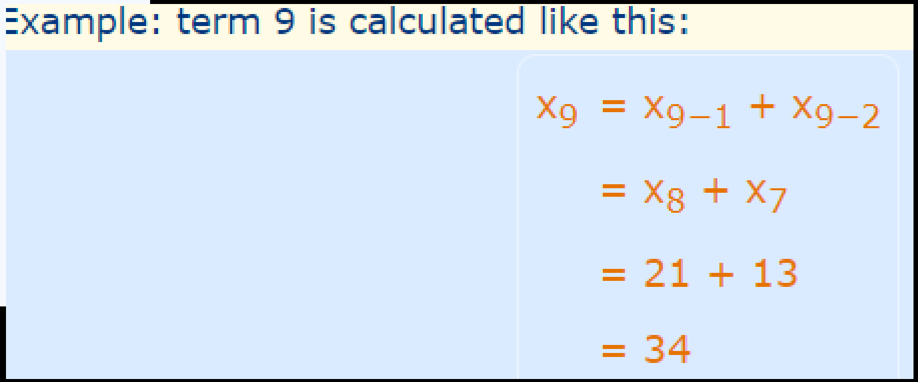
So we can write the rule:

The Rule is $\mathbf{x_n = x_{n-1} + x_{n-2}}$



where:

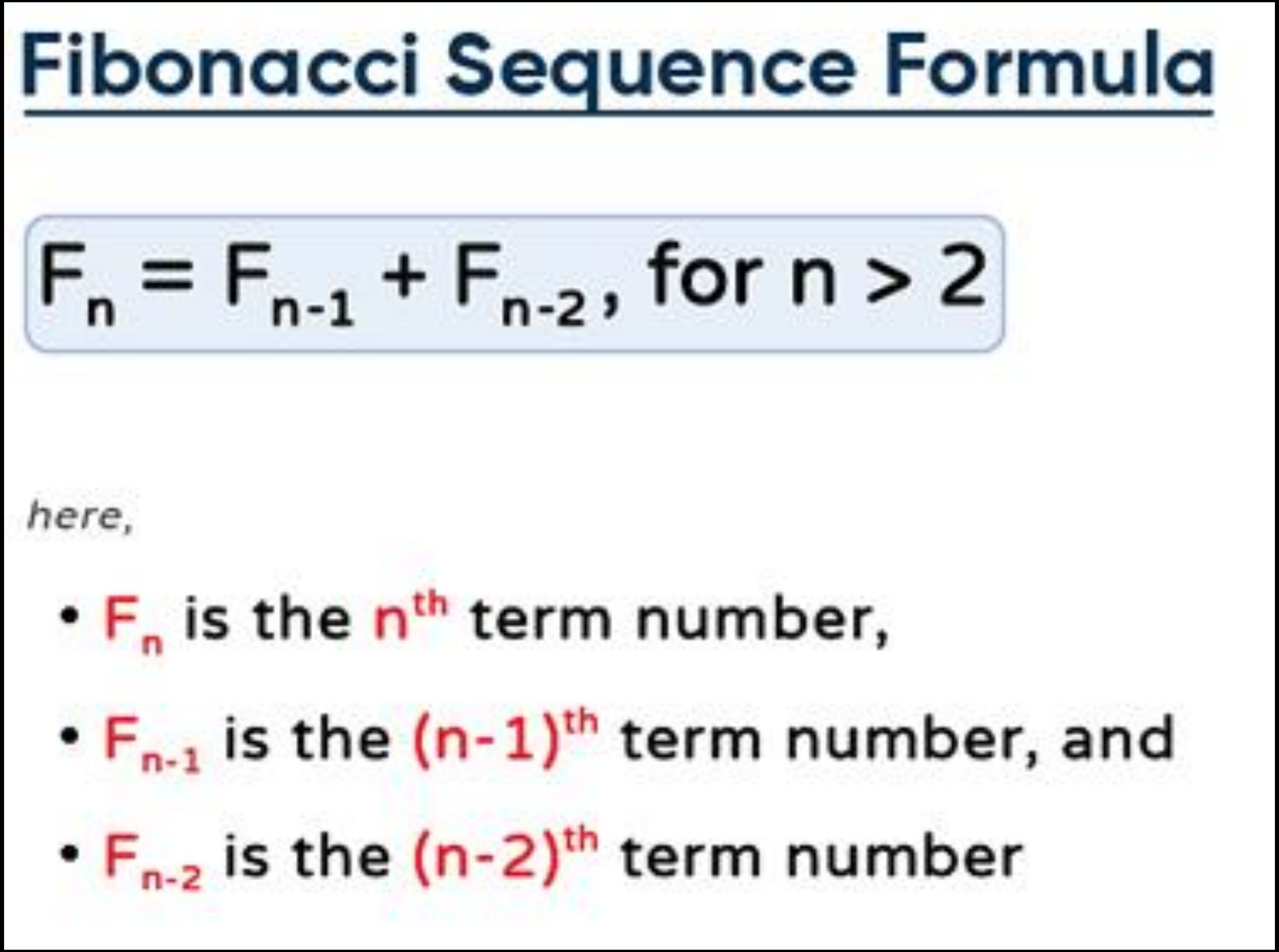
- $\mathbf{x_n}$ is term number "n"
- $\mathbf{x_{n-1}}$ is the previous term (n−1)
- $\mathbf{x_{n-2}}$ is the term before that (n−2)



Example: term 9 is calculated like this:

$$\begin{aligned} x_9 &= x_{9-1} + x_{9-2} \\ &= x_8 + x_7 \\ &= 21 + 13 \\ &= 34 \end{aligned}$$

https://www.mathsisfun.com/numbers/fibonacci-sequence.html



Fibonacci Sequence Formula

$$\boxed{F_n = F_{n-1} + F_{n-2}, \text{ for } n > 2}$$

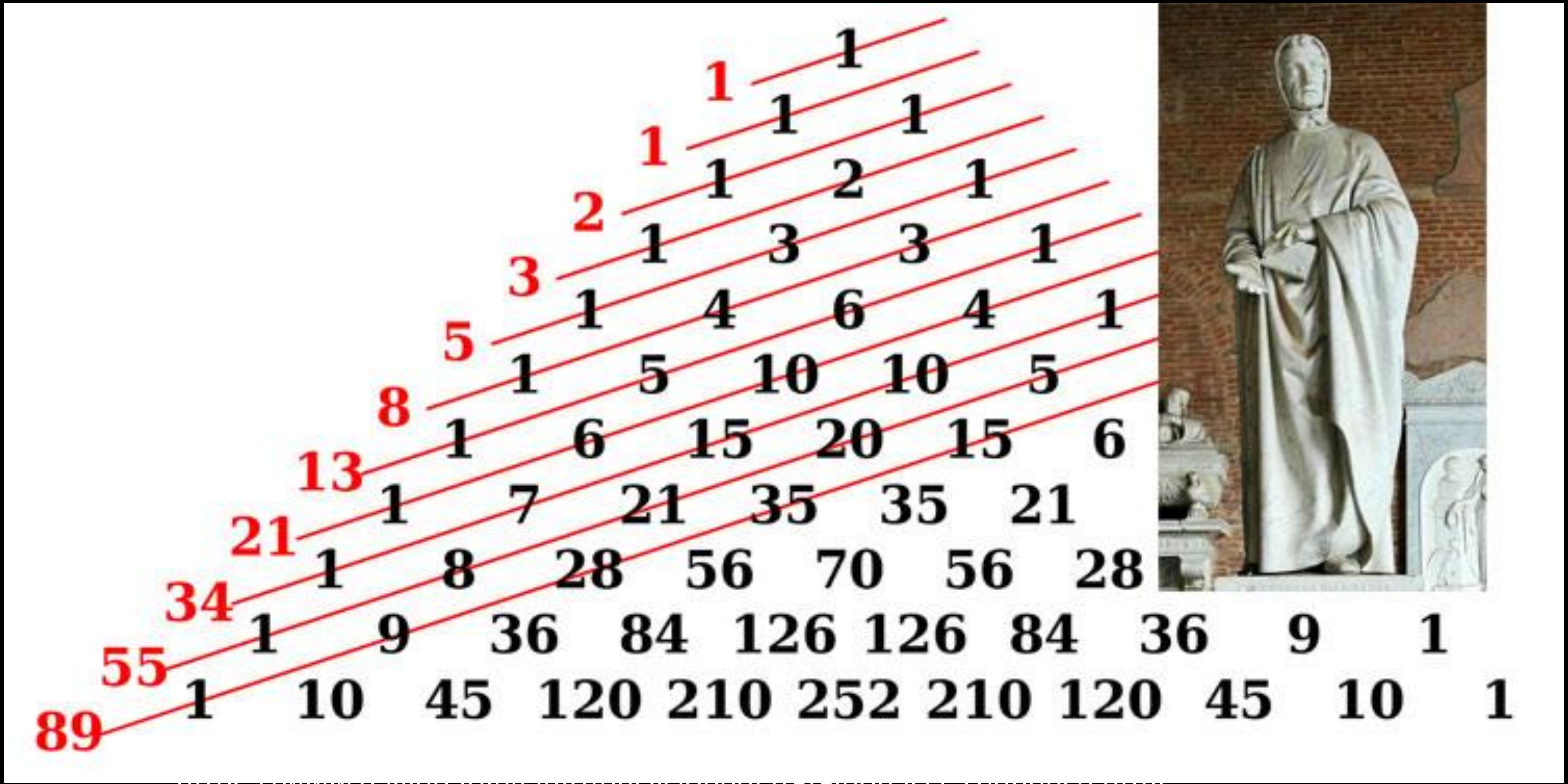
here,

- F_n is the n^{th} term number,
- F_{n-1} is the $(n-1)^{th}$ term number, and
- F_{n-2} is the $(n-2)^{th}$ term number

https://mathmonks.com/fibonacci-sequence

Fibonacci Sequence

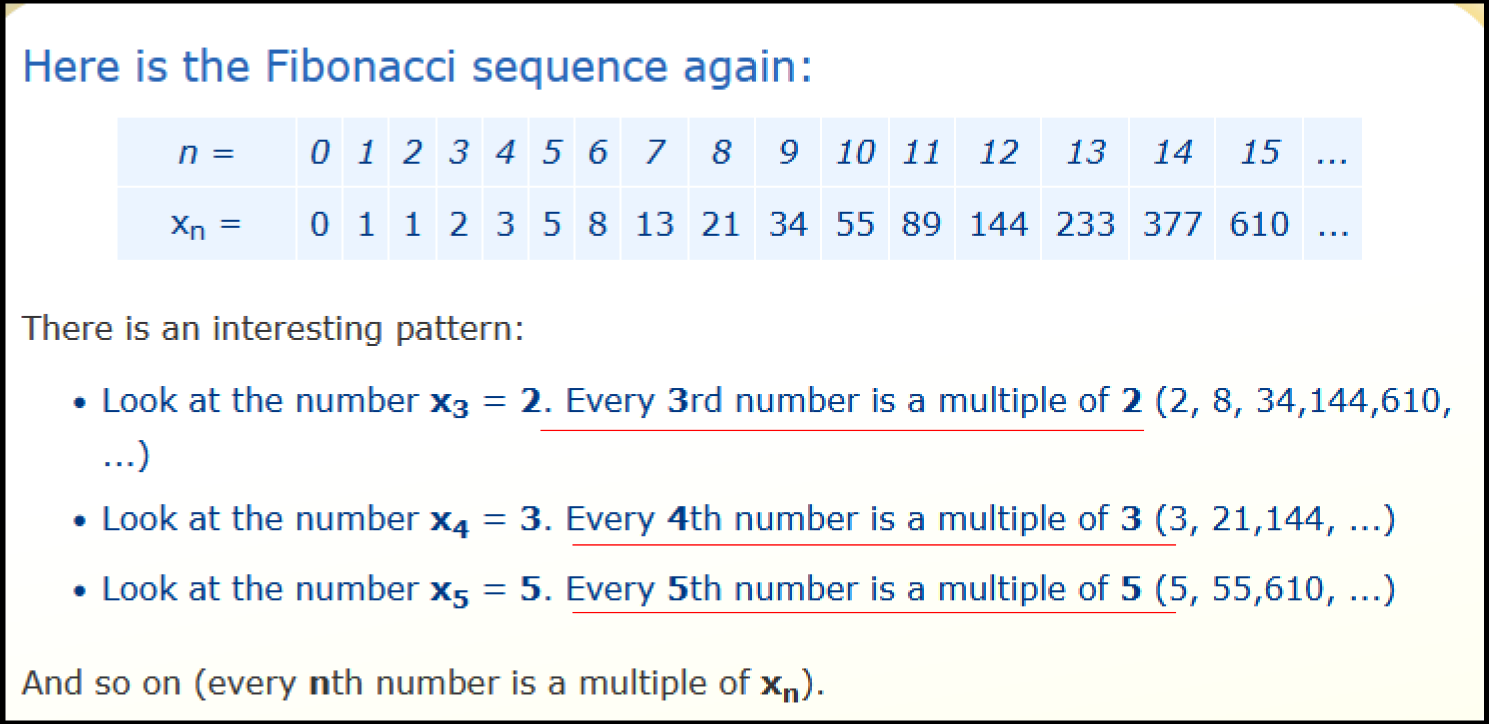
Leonardo Fibonacci. The Italian mathematician, who was bor
around A.D. 1170, was initially known as Leonardo of Pisa.



https://www.daviddarling.info/encyclopedia/F/Fibonacci_sequence.html

Fibonacci Sequence ctd..

- Here is a longer list:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,144,233,377,610,987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, ...
- *Can you figure out the next few numbers?*



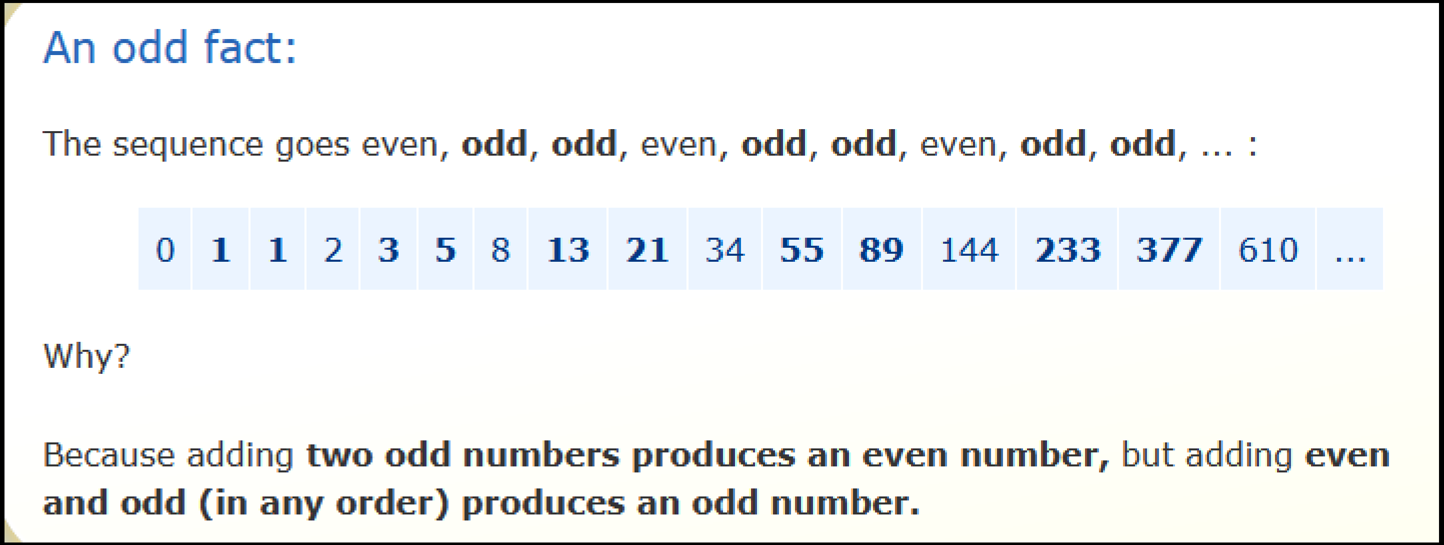
Here is the Fibonacci sequence again:

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
$x_n =$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	...

There is an interesting pattern:

- Look at the number $\mathbf{x_3}$ = **2**. Every **3**rd number is a multiple of **2** (2, 8, 34,144,610, ...)
- Look at the number $\mathbf{x_4}$ = **3**. Every **4**th number is a multiple of **3** (3, 21,144, ...)
- Look at the number $\mathbf{x_5}$ = **5**. Every **5**th number is a multiple of **5** (5, 55,610, ...)

And so on (every **n**th number is a multiple of $\mathbf{x_n}$).



An odd fact:

The sequence goes even, **odd**, **odd**, even, **odd**, **odd**, even, **odd**, **odd**, ... :

0	**1**	**1**	2	**3**	**5**	8	**13**	**21**	34	**55**	**89**	144	**233**	**377**	610	...

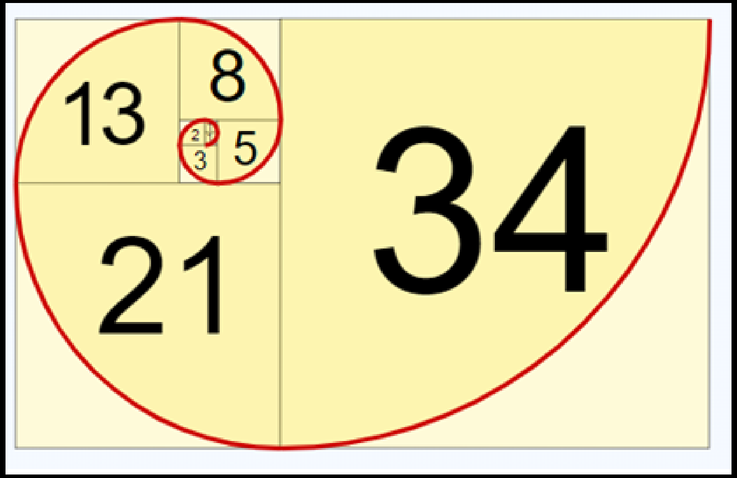
Why?

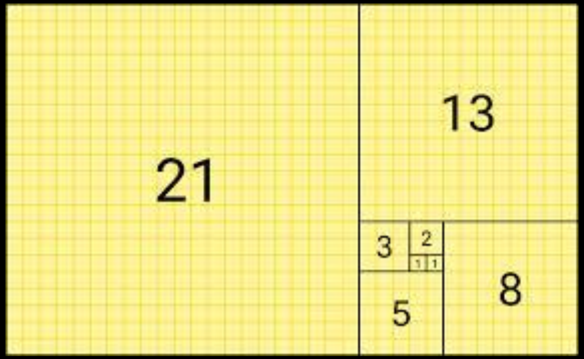
Because adding **two odd numbers produces an even number,** but adding **even and odd (in any order) produces an odd number.**

Fibonacci Sequence ctd..

When we **make squares with those widths**, we get a nice spiral:

- Phi (uppercase Φ /lowercase ϕ), is the 21st letter of the Greek alphabet, used to represent the "ph" sound in Ancient Greek. : Phi is an irrational mathematical constant, approximately 1.618.., and is often denoted by the Greek letter ϕ.
- Other commonly used names for Phi are: Golden Mean, Extreme and Mean Ratio, Divine Proportion and Golden Ratio.
- Question : (i) Does Phi ϕ a naturally occurring ratio which exhibits aesthetically pleasing properties? What is Golden Ratio?

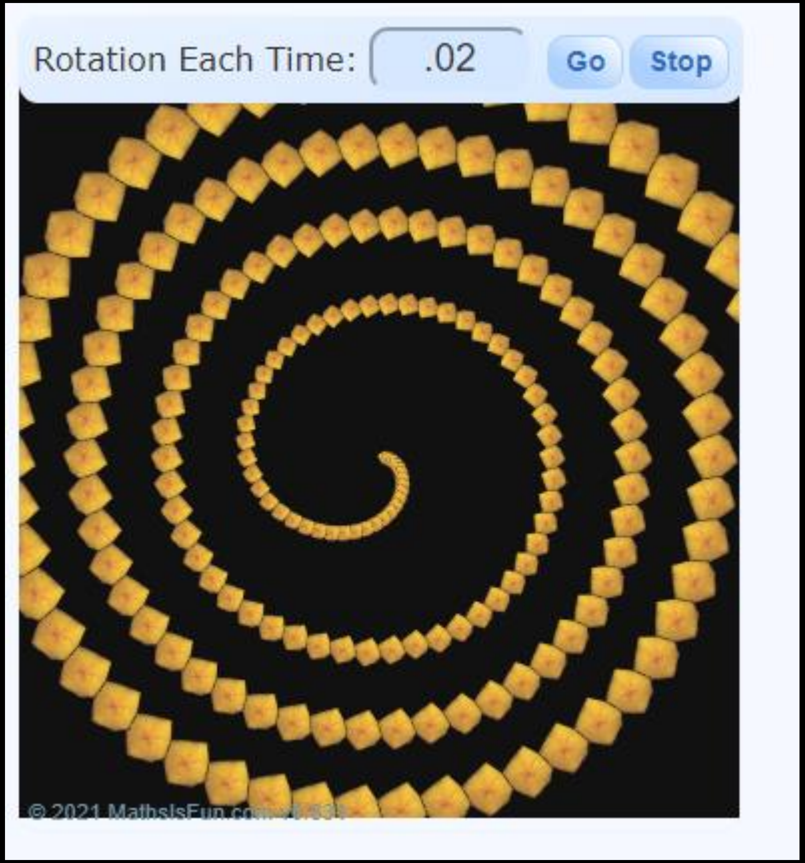


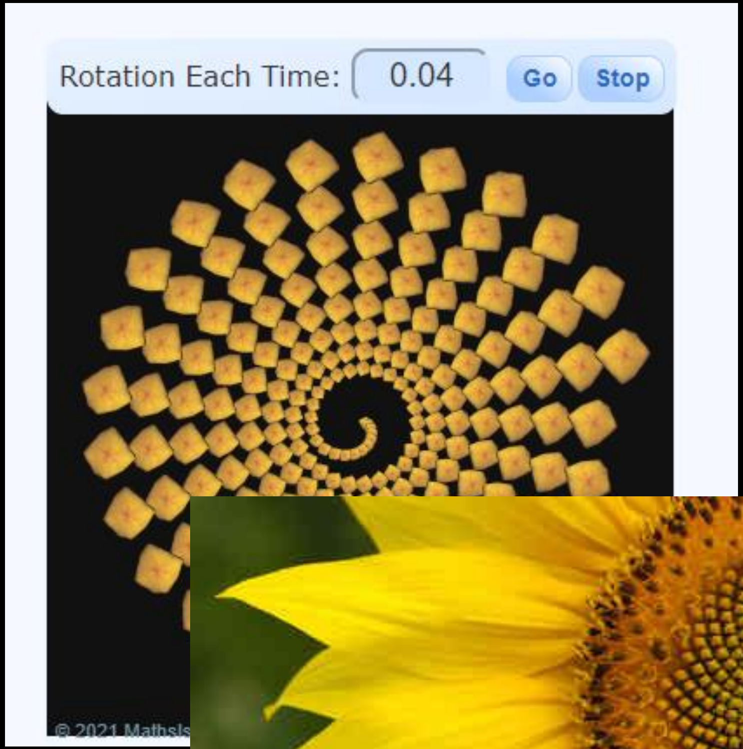


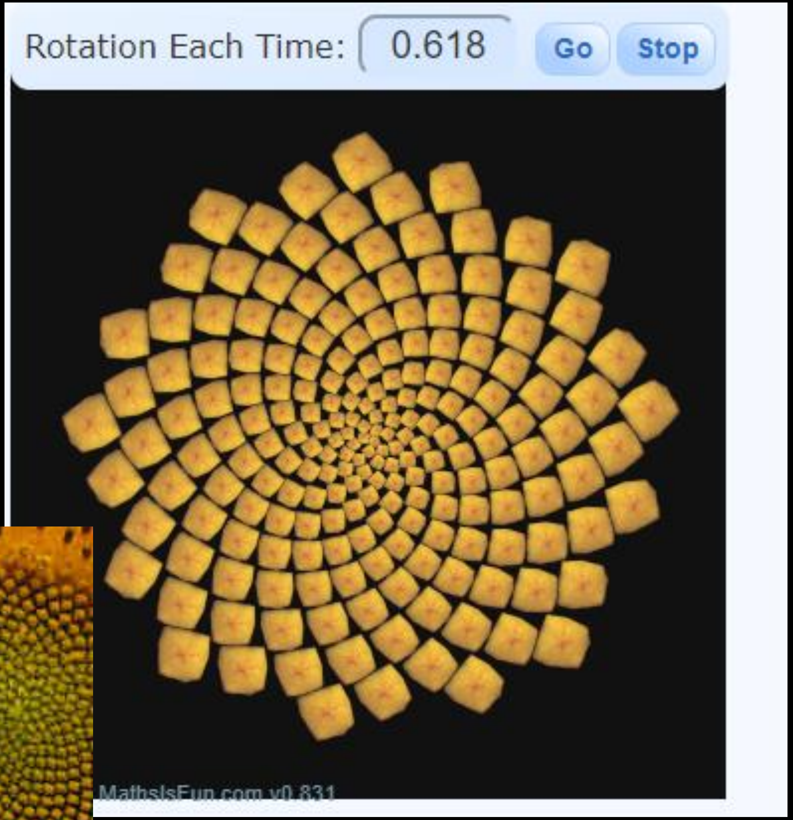
Workout activities in Golden Ratio
Refer https://www.mathsisfun.com/numbers/golden-ratio.html

Refer : Nature, The Golden Ratio, and Fibonacci too ...

https://www.mathsisfun.com/numbers/nature-golden-ratio-fibonacci.html







https://www.mathsisfun.com/numbers/fibonacci-sequence.html

Natural Phenomena and Mathematics

- This interesting behavior is not just found in sunflower seeds.
- Leaves, branches and petals can grow in spirals, too.
- Why? So that new leaves don't block the sun from older leaves, or so that the maximum amount of rain or dew gets directed down to the roots.
- **Question**: What are Golden Ratio and Golden Angle? What is the relationship between Golden Ratio and Fibonacci Sequence?

https://www.mathsisfun.com/numbers/fibonacci-sequence.html

The Fibonacci Sequence

- Problem
  - Determine the *n*-th term in the Fibonacci sequence.

$f_0 = 0$

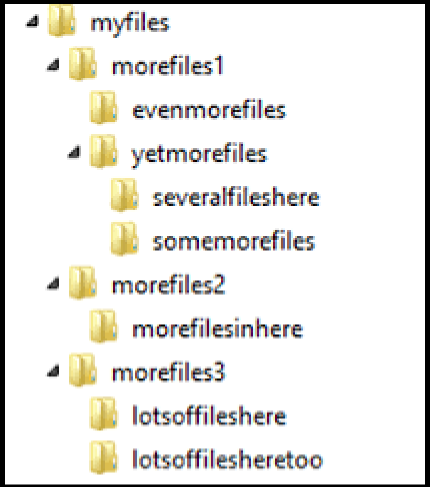
$f_1 = 1$

$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2$

- Inputs
  - A non-negative integer n.
- Outputs
  - The *n*-th term of the Fibonacci sequence.

Recursive

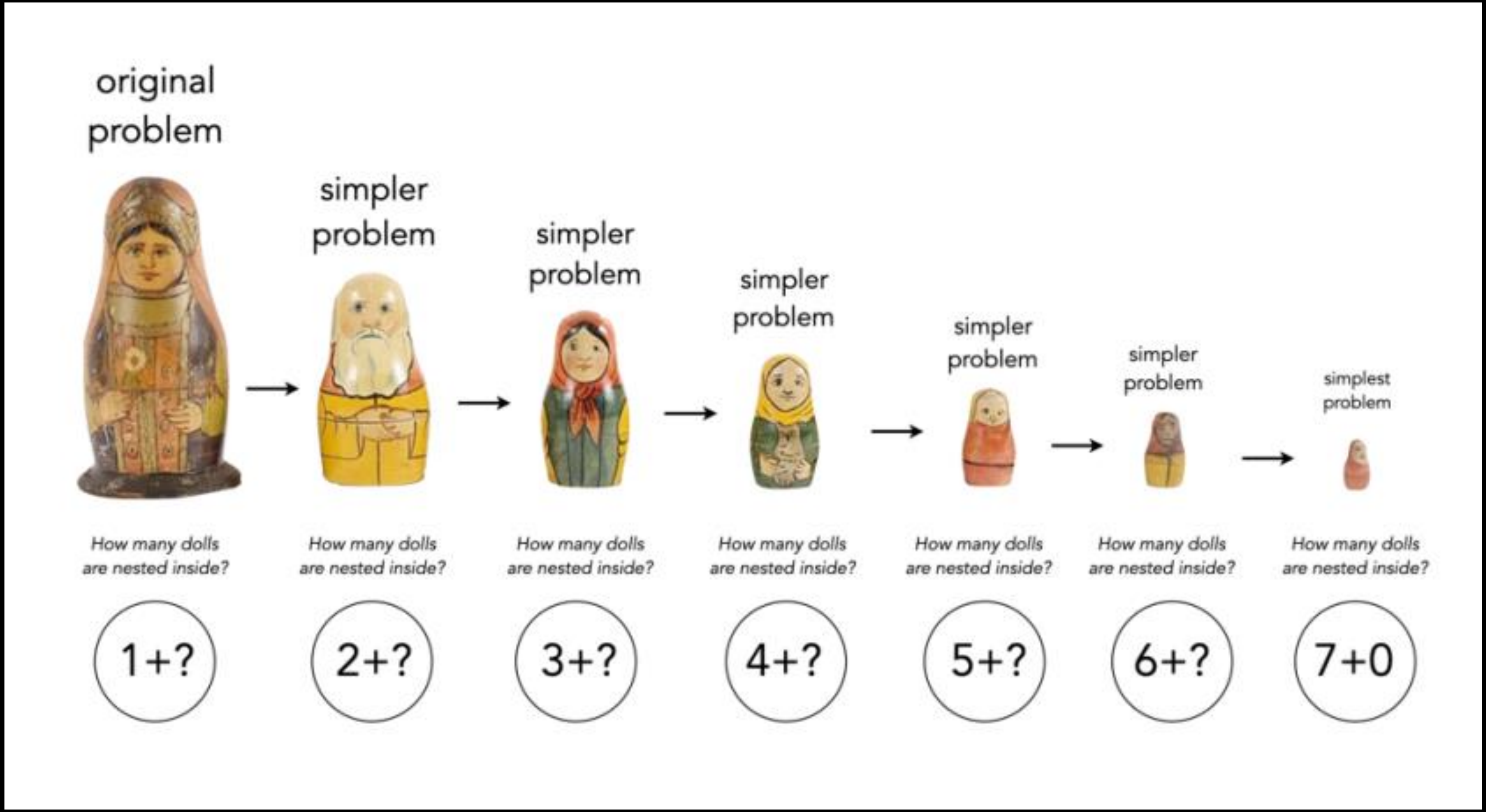
- Recurrence Relation in Mathematics?
- In English "recur" means to occur again (and again)
- Where a number is calculated from numbers before it in a sequence.
- Example: the factorial function has each number calculated from the previous number in this way:
  - n! = n(n-1)!
  - So 10! = 10 x 9!, and 9! = 9x8!, down to 1!=1
- The Fibonacci Sequence is another example.
- Another example from Data? recursive data found in every computer: its filesystem in OS.



```
myfiles
  morefiles1
    evenmorefiles
    yetmorefiles
      severalfileshere
      somemorefiles
  morefiles2
    morefilesinhere
  morefiles3
    lotsoffileshere
    lotsoffilesheretoo
```

thinking recursively?

- Another good example of thinking recursively can be found in the Russian nesting dolls, or Matryoshka dolls.
- How do determine how many dolls are inside the doll?
- The best approach is to continuously open the outer doll, add it to the count, and then solve the problem of how many dolls remain nested?
- At each opening, the problem becomes smaller, because there are fewer dolls.
- At some point the problem must be solved, because no more dolls can be nested (ignore the differences in scale of the dolls).



Recursive problem solving

- Is this recursive problem solving? In some respects yes, it is.
- Each time the outer doll is separated, the problem now becomes smaller, both figuratively and literally.
- When we reach the doll which can not be separated, the problem is solved.
- Here 👉 is how is could be described algorithmically (in a round-about sort of way):

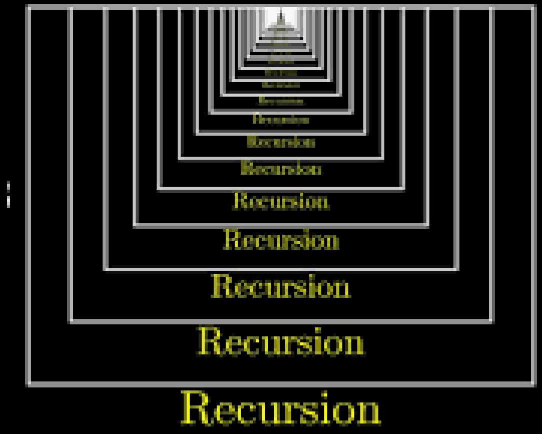
```
doll-count is 0

count-NestingDolls(doll)
begin
   if (doll can be separated)
      smaller-doll = separate(doll)
      increase doll-count by 1
      perform count-NestingDolls(smaller-doll)
   otherwise
      increase doll-count by 1
      done
end
```

Ex. Write the Loop Version of the above Algorithm(i.e no recursion)

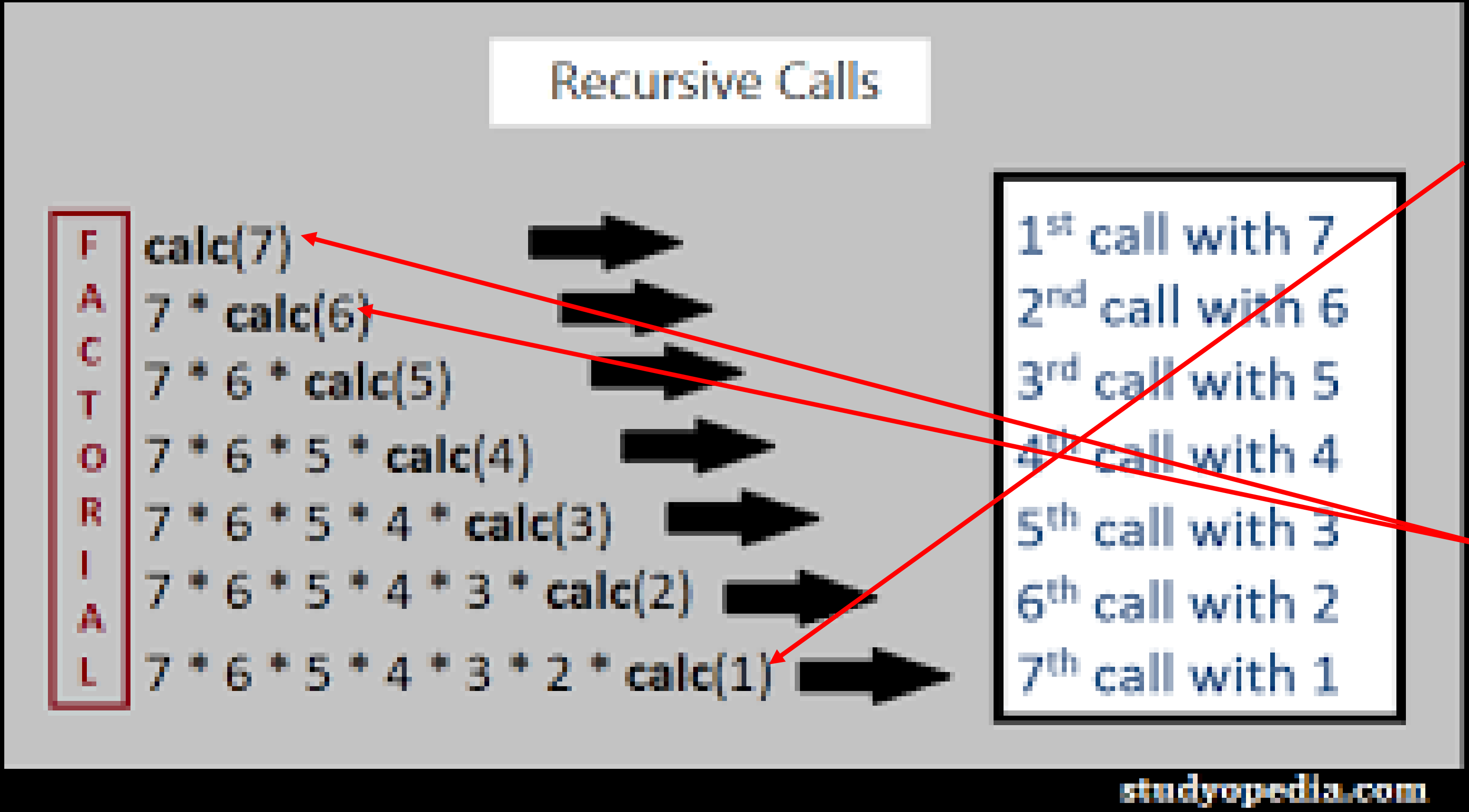
Recursive?

- Applying a rule or formula to its own result, again and again.
- Example: 1, 2, 4, 8, 16, 32, ...
-
- **The base case**
  - You might ask, ``Couldn't a recursive algorithm go on forever?''
  - Every recursive algorithm *must* have a case in which it does not recurse -- called the **base case**.



https://www.mathsisfun.com/definitions/recursive.html

Structure of Recursive Implementations



- A recursive implementation always has <u>two parts</u>:
- base case, which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
- recursive step, which decomposes a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then recombines the results of those subproblems to produce the solution to the original problem.

Refer: **Reading 10: Recursion**

https://web.mit.edu/6.005/www/fa15/classes/10-recursion/#:~:text=implementation%20can%20solve.-,Common%20Mistakes%20in%20Recursive%20Implementations,the%20base%20cases%20are%20covered.

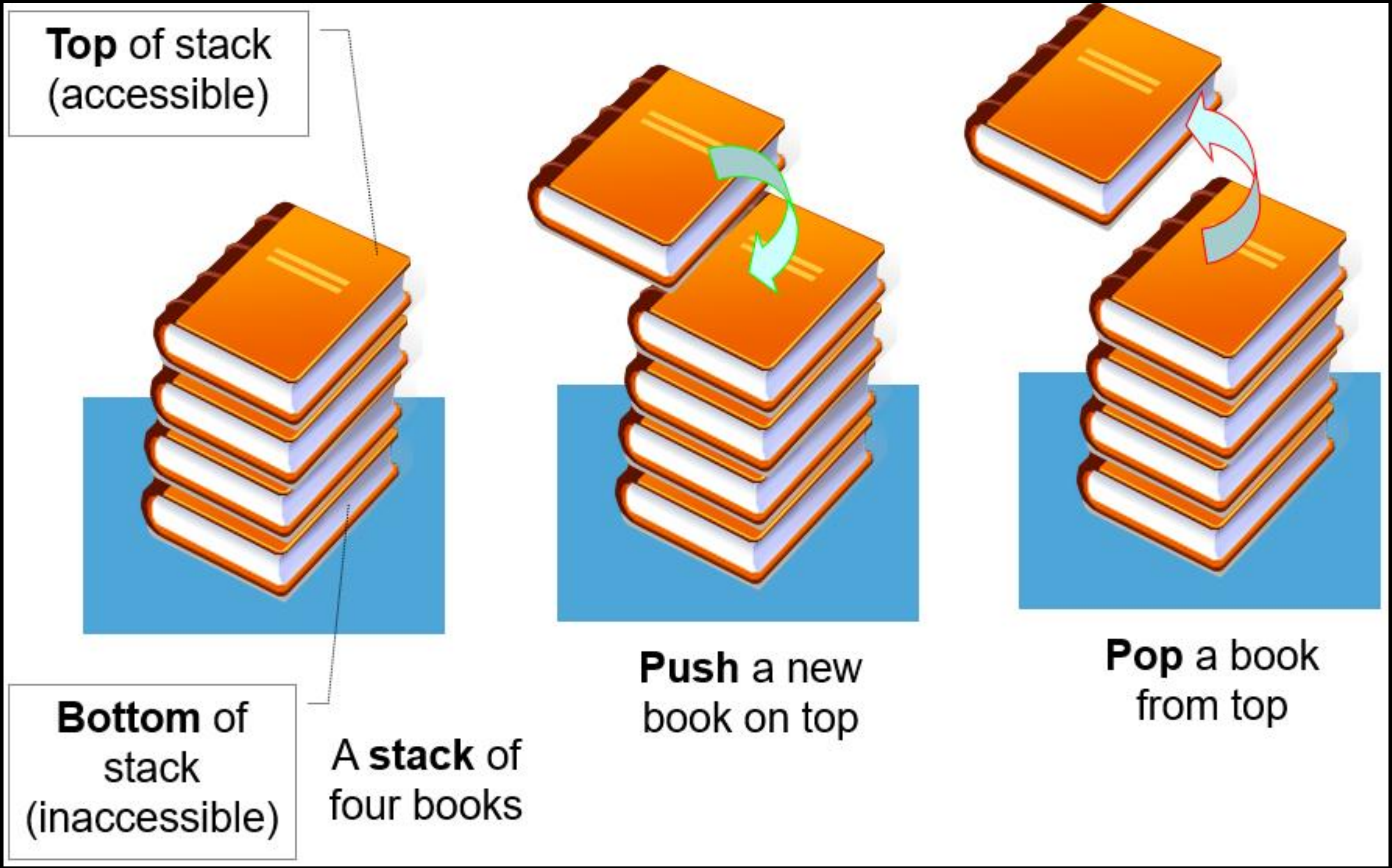
Properties of Recursion:

- Recursion has some important properties. Some of which are mentioned below:

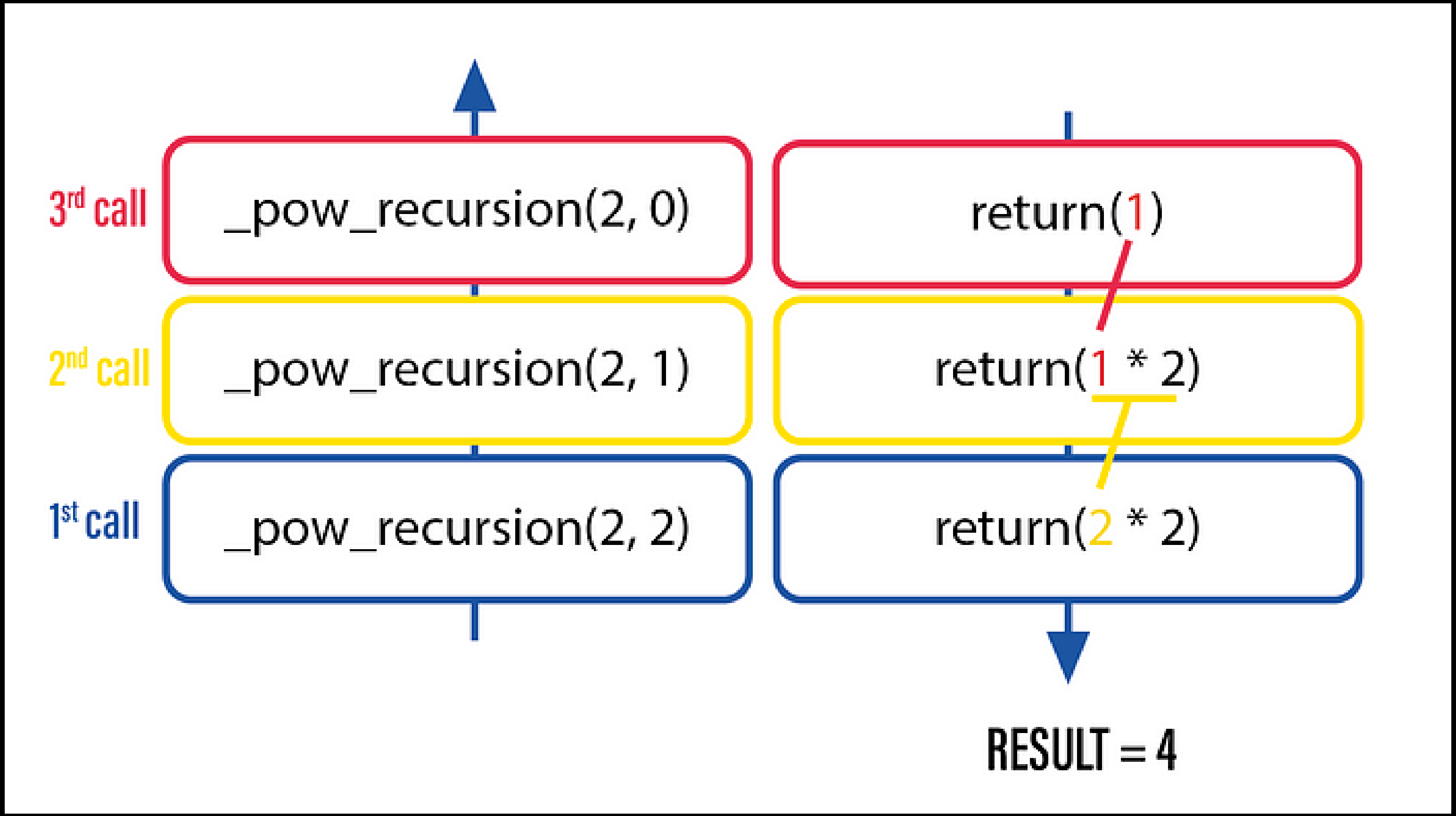
1. The primary property of recursion is the ability to solve a problem by breaking it down into smaller sub-problems, each of which can be solved in the same way.
2. A recursive function must have a base case or stopping criteria to avoid infinite recursion.
3. Recursion involves calling the same function within itself, which leads to a call stack.
4. Recursive functions may be less efficient than iterative solutions in terms of memory and performance.

https://www.geeksforgeeks.org/what-is-recursion/

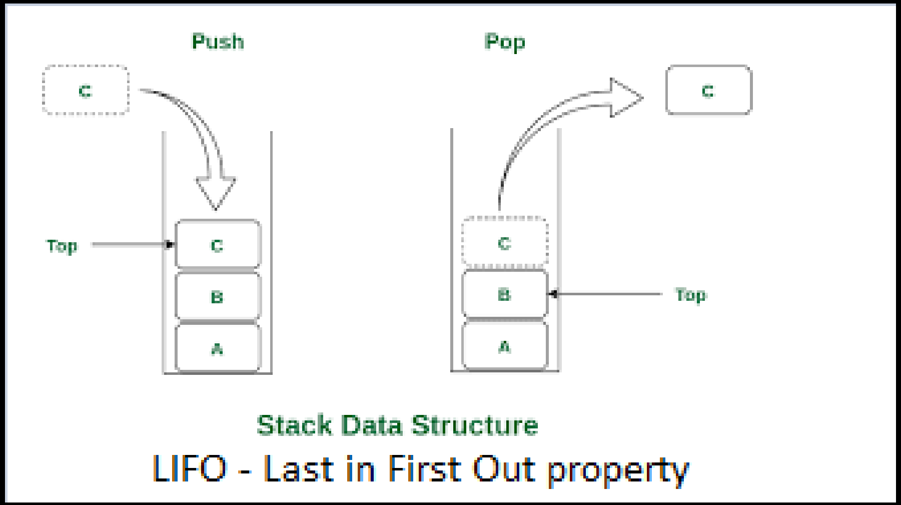
Stack?


Top of stack (accessible)
Bottom of stack (inaccessible)
A **stack** of four books
Push a new book on top
Pop a book from top

Recursion managed with a 'Stack'?

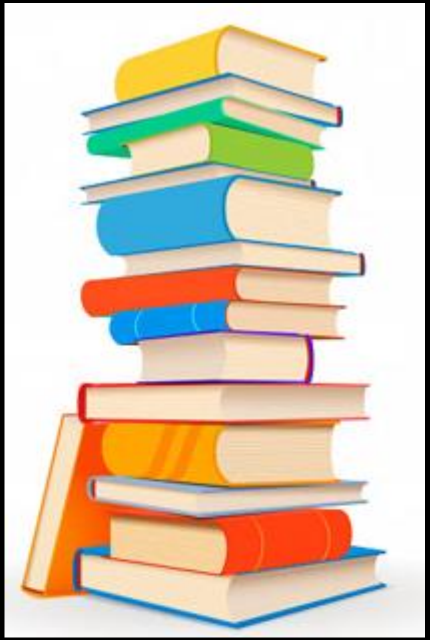


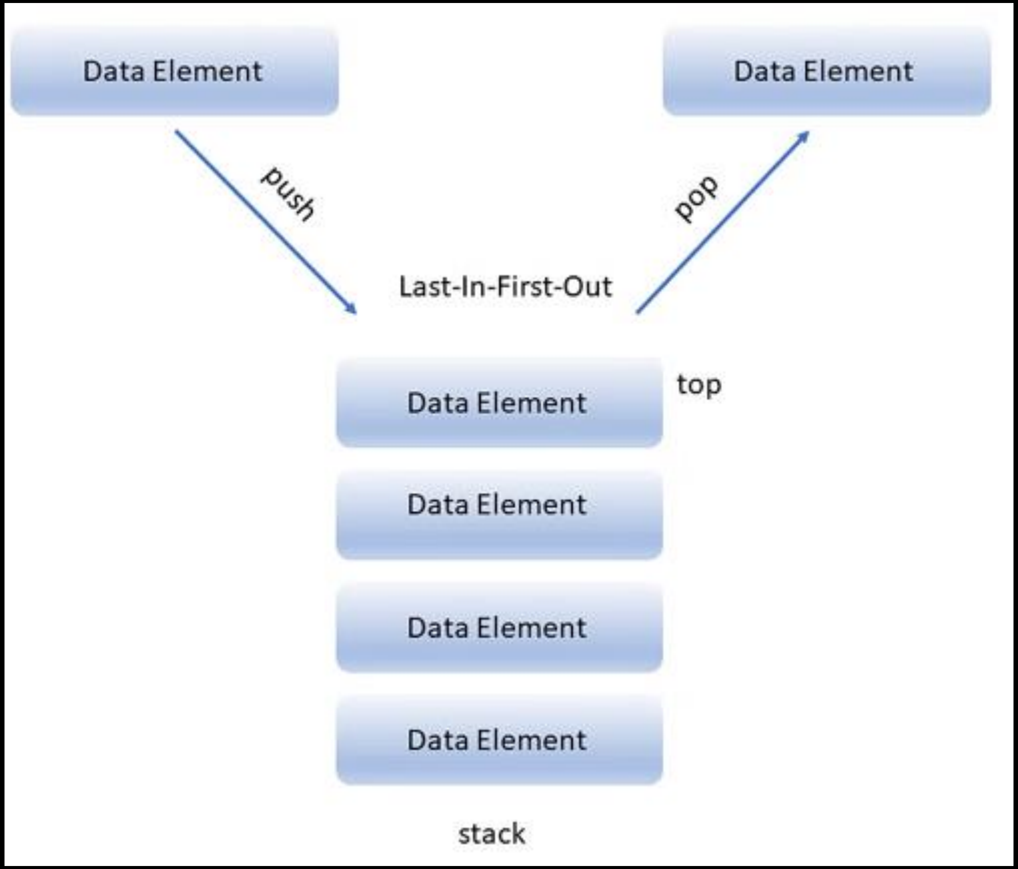
What is a Stack Data Structure?

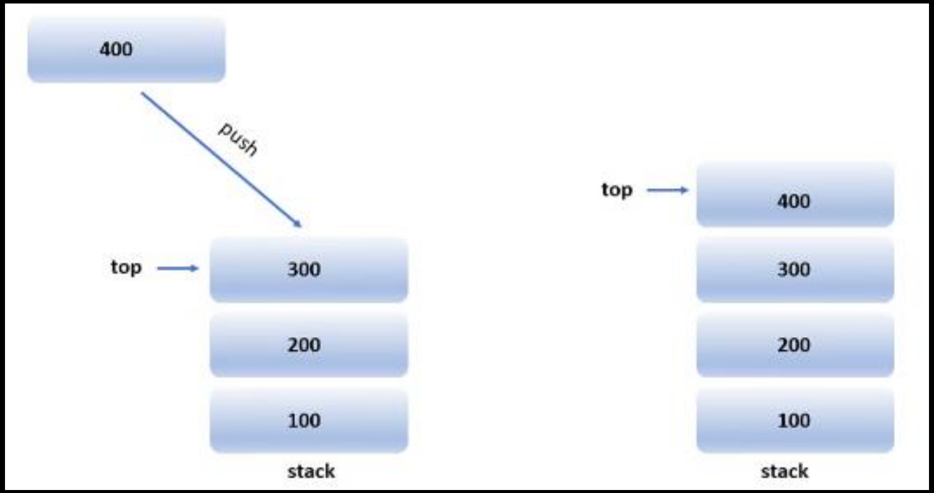


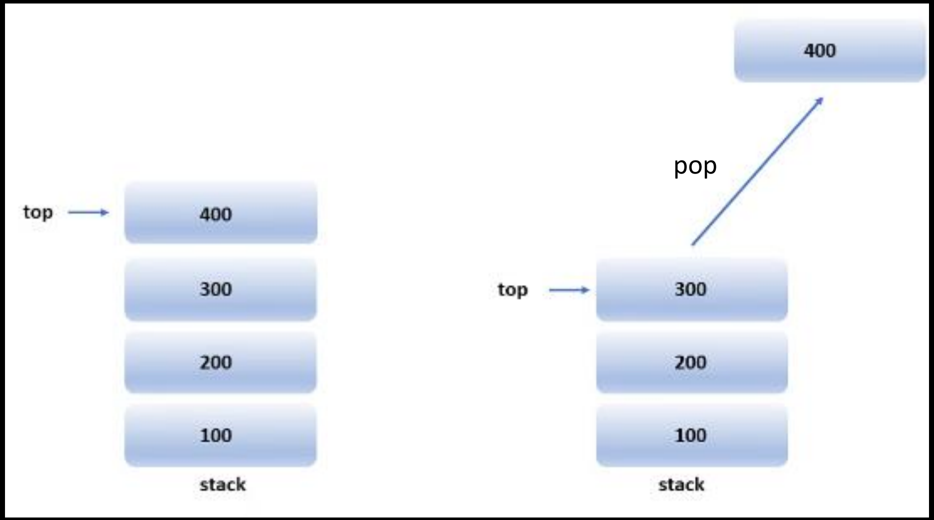
https://medium.com/@4318_26766/recursion-and-how-it-works-on-the-stack-bdcdce726331

Stack with Last in First Out(LIFO) property?





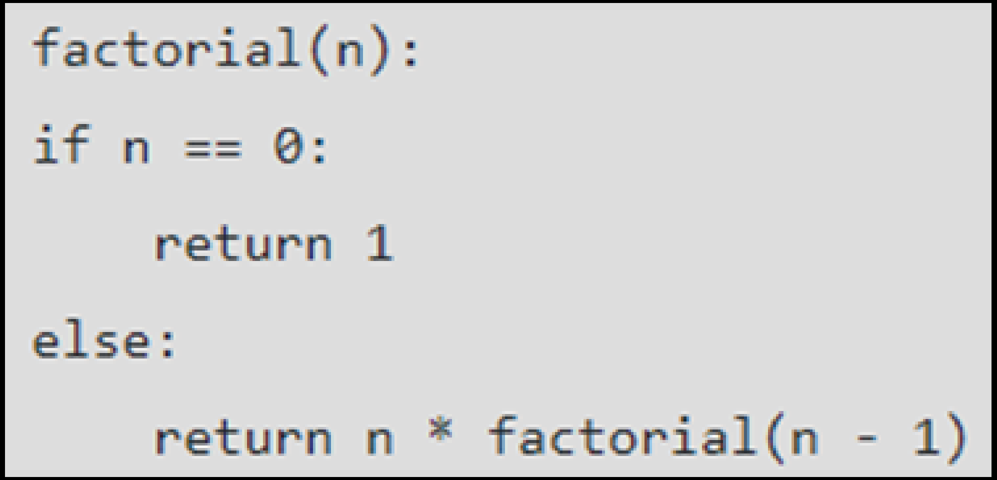




Refer Stack in Data Structure: Master LIFO Concepts Easily
https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures

Types of Recursion: Direct recursion

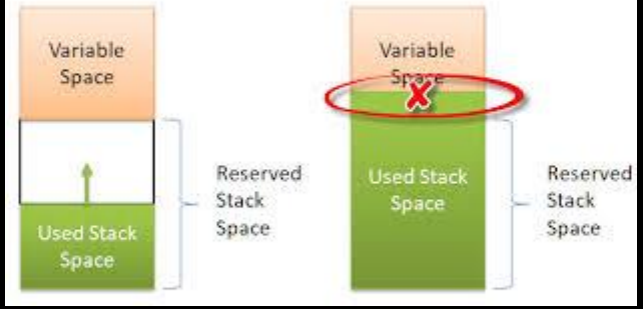
- Direct recursion is a type of recursion in which a function directly calls itself during its execution.
- The function solves a smaller subproblem and then calls itself with the reduced subproblem until it reaches a base case that terminates the recursion.
- Direct recursion involves a straightforward and explicit self-reference within the function's body.



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

the `factorial()` function calls itself with a smaller value `n – 1` until it reaches the base case where `n` is equal to 0.
This recursive approach calculates the factorial of a given number by multiplying it with the factorial of the preceding number.

Direct Recursion - Pros and Cons:



offers some advantages in problem-solving and algorithm design:

- Simplicity: Direct recursion often provides a straightforward and intuitive solution for problems that exhibit self-similar subproblems.
- Readability: Recursive functions can often express the problem-solving logic more clearly and concisely, making the code easier to understand.

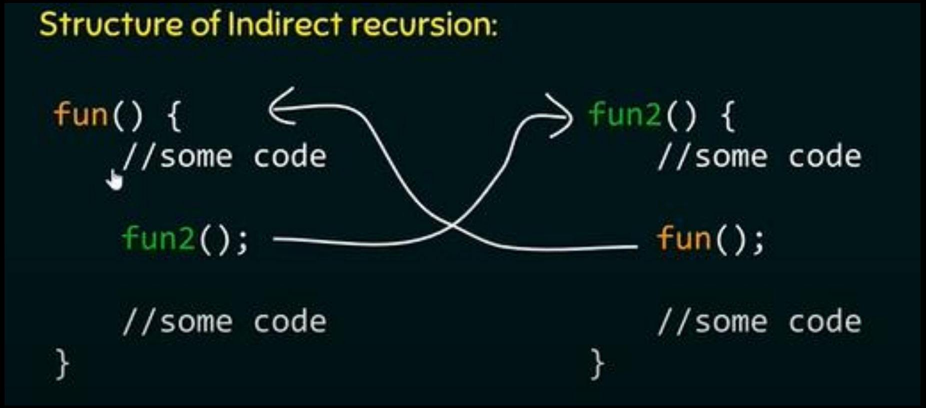
However, also has some drawbacks:

- Memory Usage: Recursive function calls consume memory as each call creates a new stack frame. If the recursion depth is large, it may lead to stack overflow errors.
- Performance Overhead: Recursive calls involve function call overhead, which can impact performance compared to iterative approaches.
- Tail Recursion (covered in this note later) Optimization: Direct recursion may not benefit from tail recursion optimization, where the recursive call is the last operation in the function. This optimization eliminates the need for maintaining multiple stack frames, enhancing performance.

https://intellipaat.com/blog/recursion-in-data-structure/

Types of Recursion: Indirect Recursion

- Indirect recursion is a type of recursion in which a function calls another function(s).
- The chain of function calls leads back to the original function, creating a cycle.
- In indirect recursion, there is a circular dependency among multiple functions, where each function calls another function(s) in a sequence until the base case is reached.


Structure of Indirect recursion:

```
fun() {                        fun2() {
    //some code                    //some code

    fun2();                        fun();

    //some code                    //some code
}                              }
```

Example Implementation:

```
void function1(int n);
void function2(int n) {
    if (n > 0) {
        cout << n << " ";
        function1(n - 1);
    }
}
void function1(int n) {
    if (n > 1) {
        cout << n << " ";
        function2(n / 2);
    }
}
int main() {
    function1(20);
    return 0;
}
```

https://intellipaat.com/blog/recursion-in-data-structure/

Indirect Recursion - **Pros and Cons:**

offers certain advantages :

- **Problem Decomposition:** Indirect recursion can be useful for breaking down a complex problem into smaller, interdependent subproblems. Each function focuses on solving a specific part of the problem.
- **Code Modularity:** By dividing the problem-solving logic across multiple functions, the code can be organized and modularized, improving readability and maintainability.

also has some drawbacks:

- **Complexity:** Indirect recursion can introduce additional complexity due to the interdependencies between functions. This complexity can make code harder to understand and debug.
- **Execution Order:** The execution order of functions in indirect recursion is crucial. Incorrect sequencing or missing base cases can lead to infinite loops or incorrect results.
- **Performance Overhead:** Similar to direct recursion, indirect recursion can incur function call overhead and memory consumption. Care must be taken to avoid excessive recursive calls.

https://intellipaat.com/blog/recursion-in-data-structure/

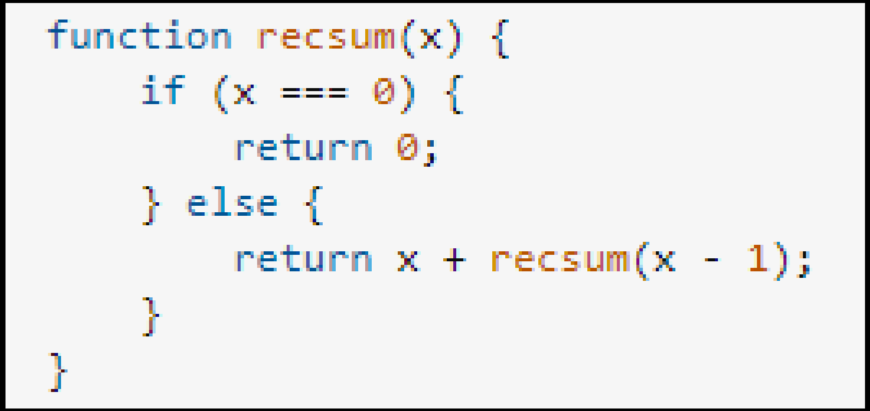
Types of Recursion:

1. Direct recursion: When a function is called within itself directly it is called direct recursion. This can be further categorized into four types:
   i. Tail recursion,
   ii. Head recursion,
   iii. Tree recursion and
   iv. Nested recursion.
2. Indirect recursion: Indirect recursion occurs when a function calls another function that eventually calls the original function and it forms a cycle.

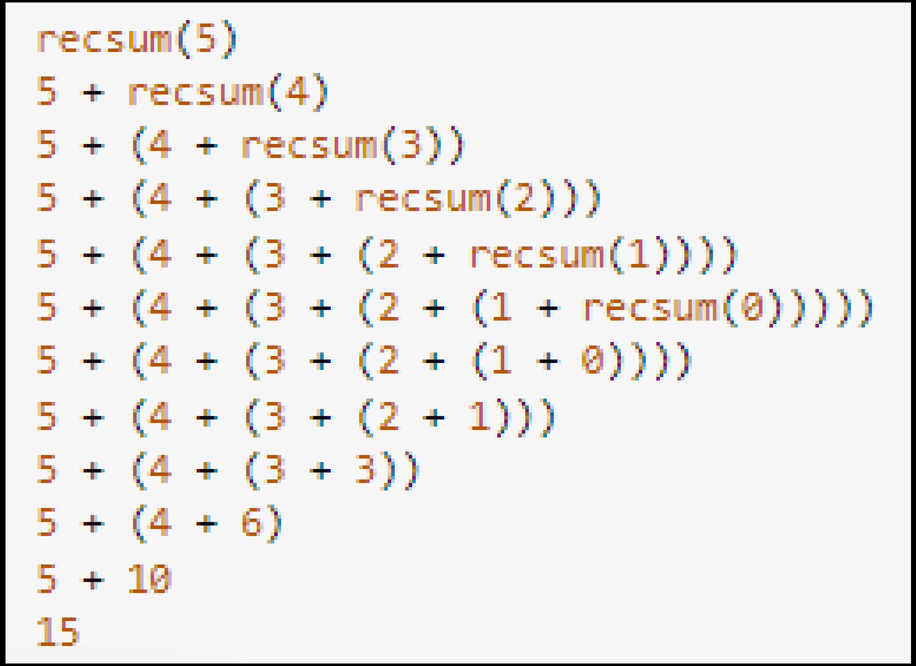
Consider a simple function that adds the first N natural numbers. (e.g. sum(5) = 0 + 1 + 2 + 3 + 4 + 5 = 15).

Tail Recursion

- Tail recursion is defined as a recursive function in which the recursive call is the **last statement** that is executed by the function. Hence nothing is left to execute after the recursion call.
  - e.g Consider a simple function that adds the first N natural numbers. (e.g. sum(5) = 0 + 1 + 2 + 3 + 4 + 5 = 15).



```
function recsum(x) {
    if (x === 0) {
        return 0;
    } else {
        return x + recsum(x - 1);
    }
}
```

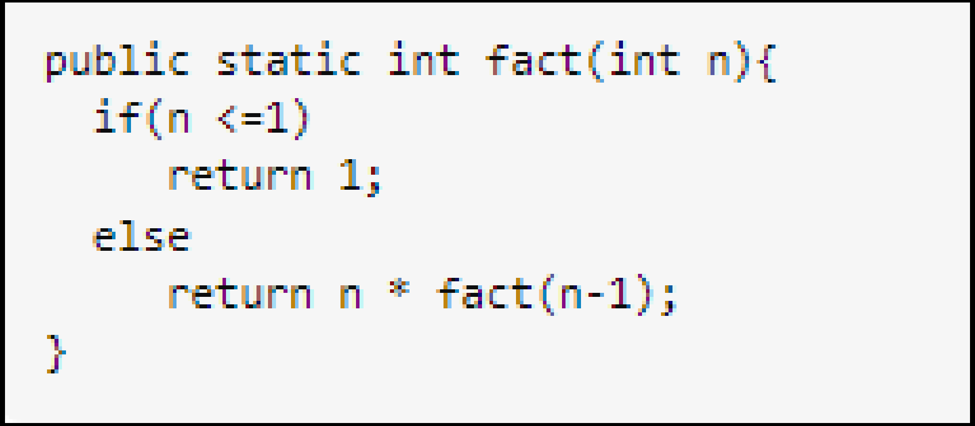


```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + (1 + recsum(0)))))
5 + (4 + (3 + (2 + (1 + 0))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

https://stackoverflow.com/questions/33923/what-is-tail-recursion

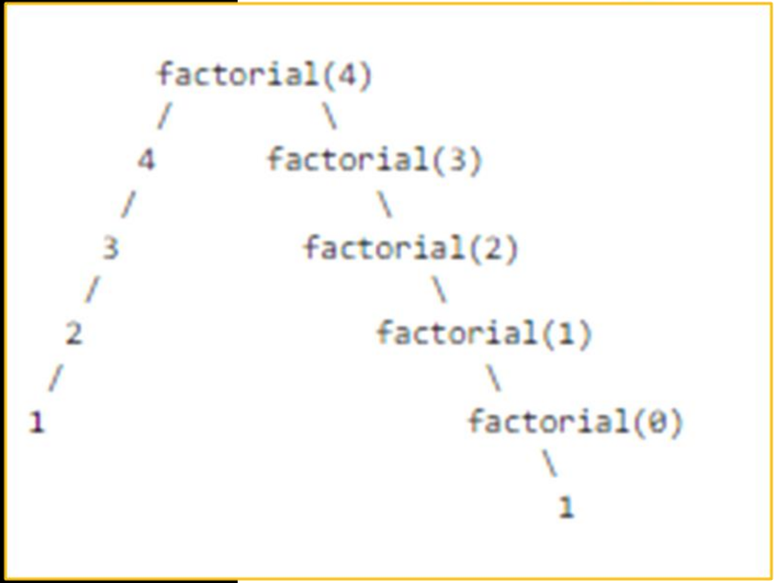
e.g. Tail Recursion – Factorial

-
- **fact(4) = 4 * 3 * 2 * 1 = 24**

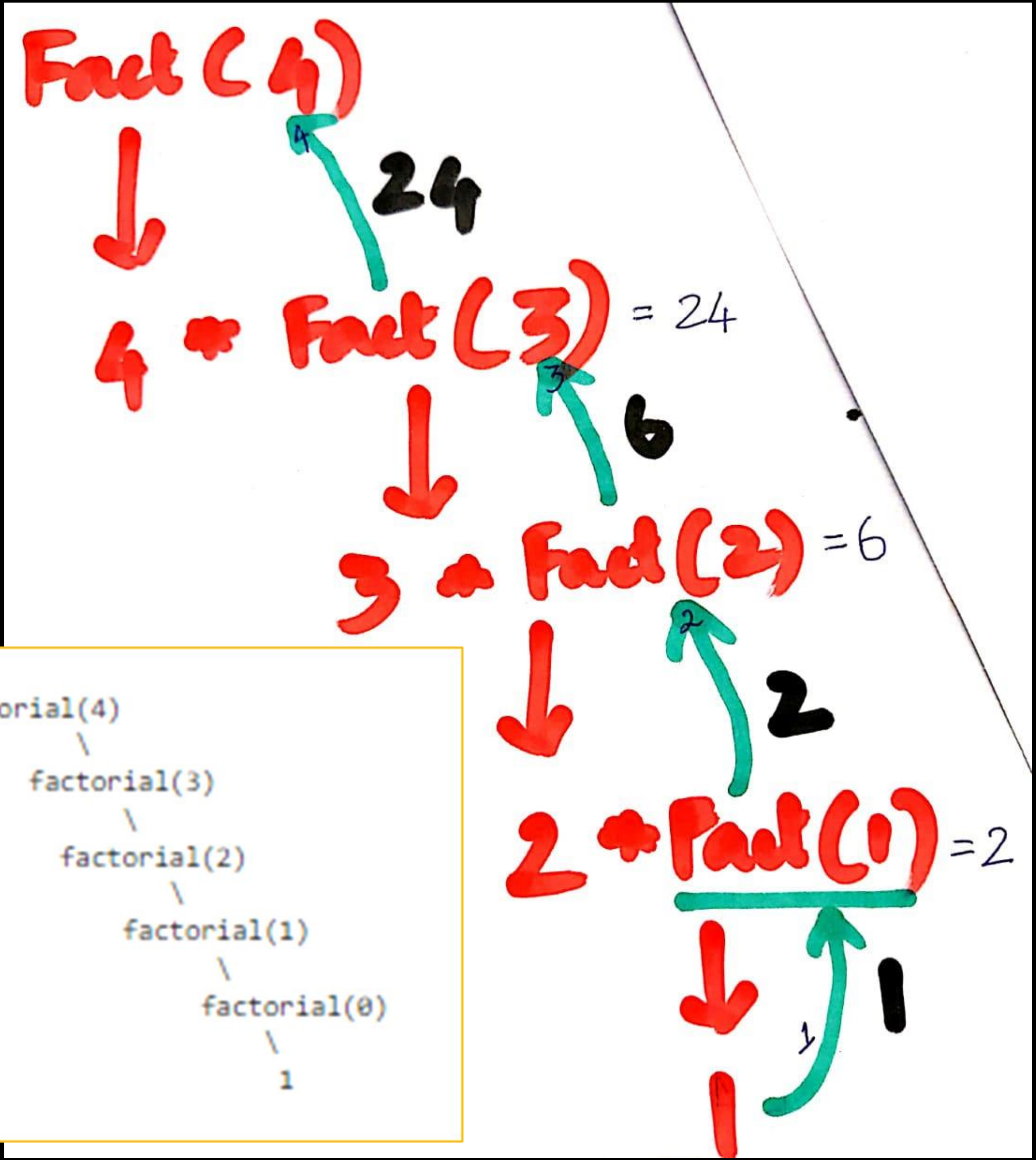


```
public static int fact(int n){
  if(n <=1)
     return 1;
  else
     return n * fact(n-1);
}
```

https://stackoverflow.com/questions/33923/what-is-tail-recursion

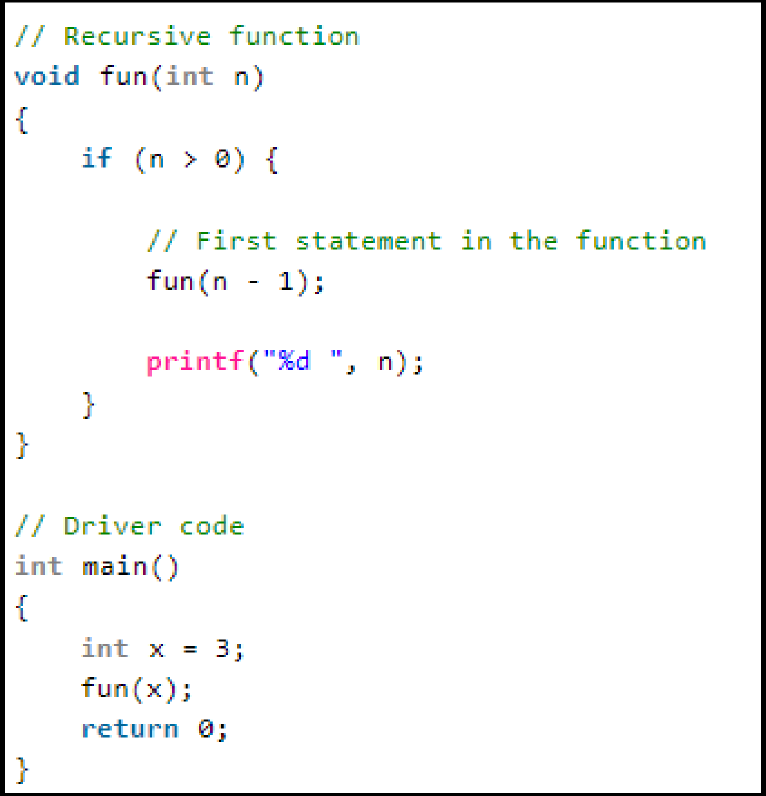


```
         factorial(4)
        /          \
       4       factorial(3)
      /              \
     3            factorial(2)
    /                  \
   2                 factorial(1)
  /                        \
 1                       factorial(0)
                               \
                                1
```



Head Recursion

- Head Recursion: If a recursive function calling itself and that recursive call is the first statement in the function then it's known as Head Recursion. There is no statement, no operation before the call.
- The function does not have to process or perform any operation at the time of calling and all operations are done at returning time.

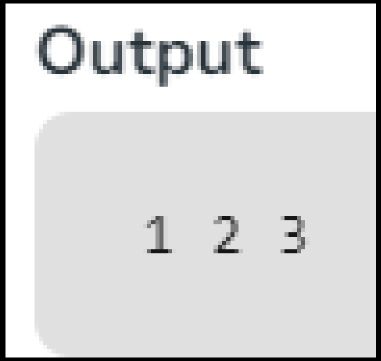


```
// Recursive function
void fun(int n)
{
    if (n > 0) {

        // First statement in the function
        fun(n - 1);

        printf("%d ", n);
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```



Output

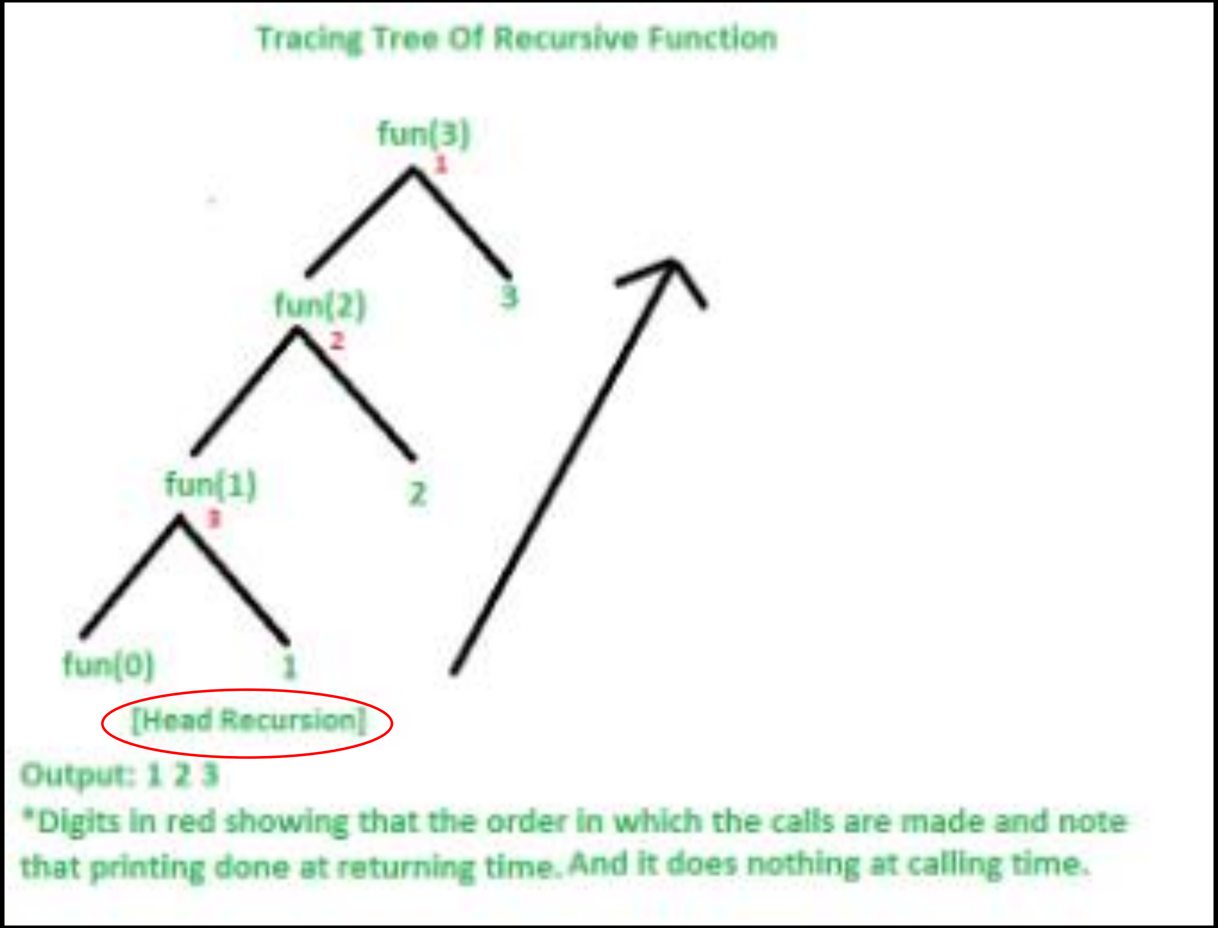
```
1 2 3
```

Refer: https://www.youtube.com/watch?v=wRdM1Z-A35Y

https://www.geeksforgeeks.org/types-of-recursions/

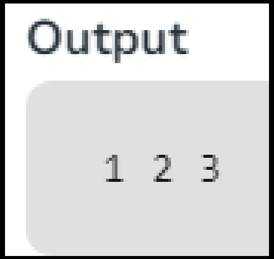
Head Recursion

- Let's understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are prod



Output: 1 2 3

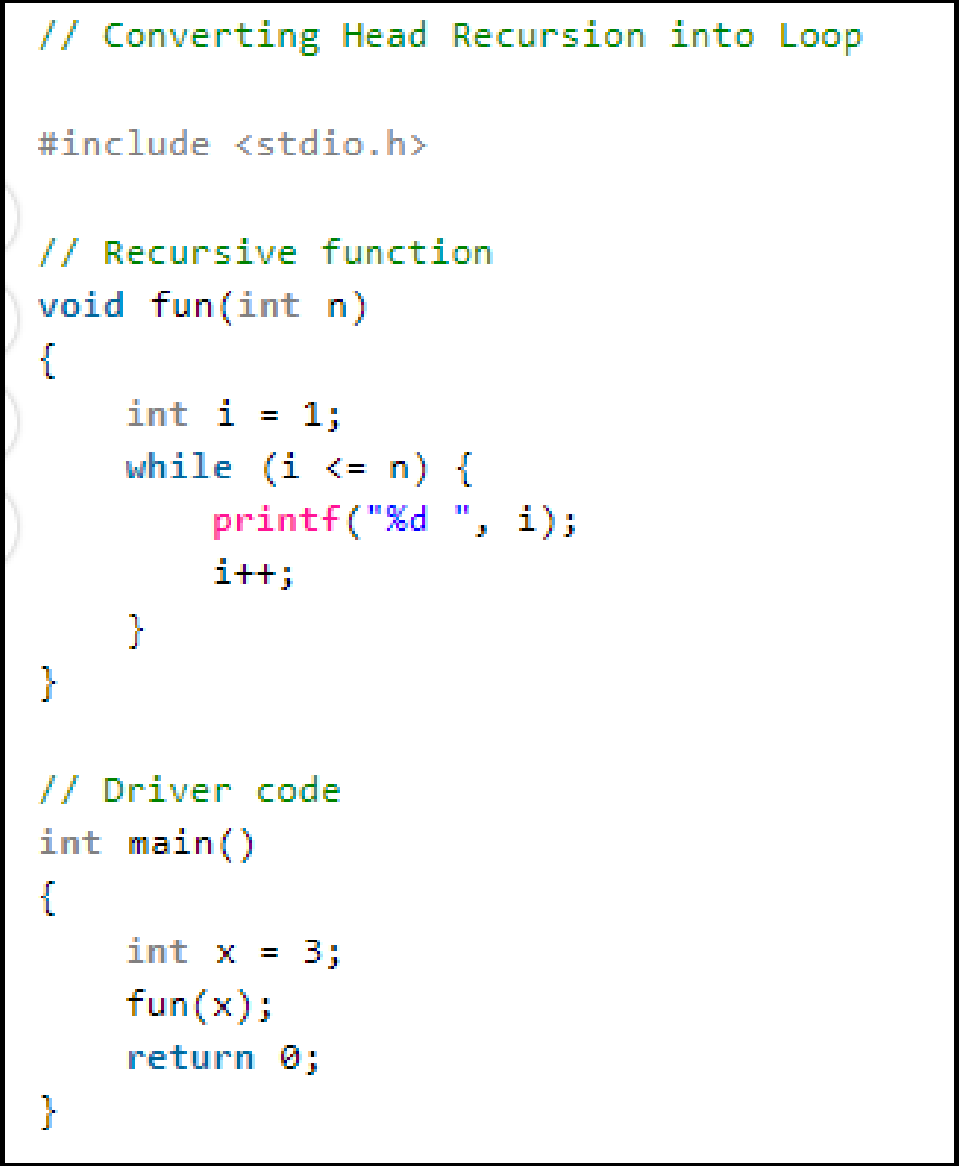
*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.



Output

```
1 2 3
```

Head Recursion : Loop version



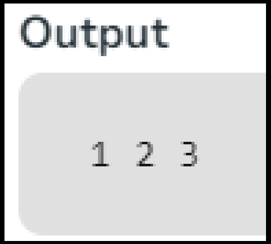
```c
// Converting Head Recursion into Loop

#include <stdio.h>

// Recursive function
void fun(int n)
{
    int i = 1;
    while (i <= n) {
        printf("%d ", i);
        i++;
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

- Loop version of Head Recursion

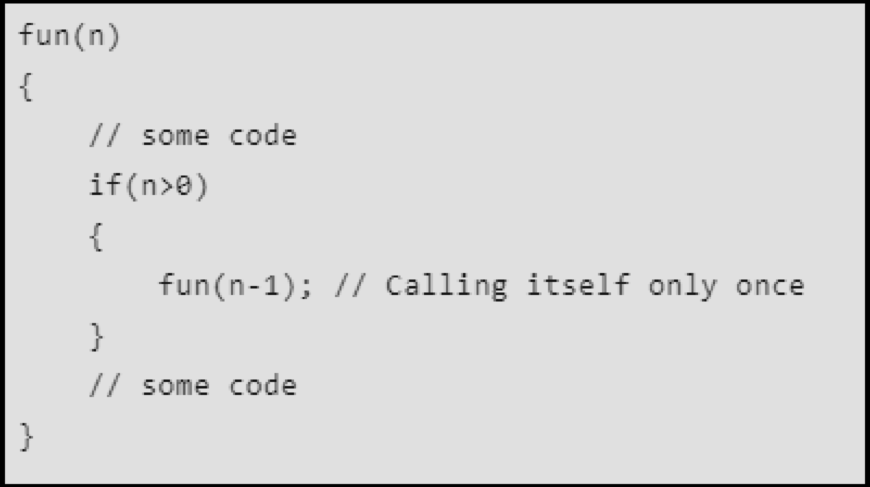


Output

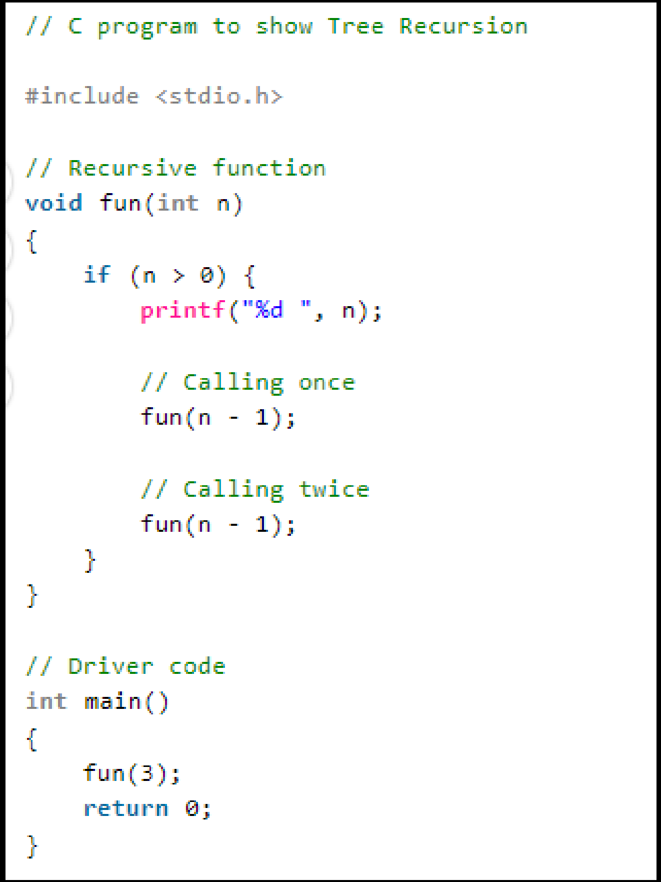
```
1 2 3
```

Tree Recursion

- Tree Recursion: To understand Tree Recursion let's first understand Linear Recursion. If a recursive function calling itself for one time then it is known as Linear Recursion. Otherwise if a recursive function calling itself for more than one time then it's known as Tree Recursion.



```
fun(n)
{
    // some code
    if(n>0)
    {
        fun(n-1); // Calling itself only once
    }
    // some code
}
```



```c
// C program to show Tree Recursion

#include <stdio.h>

// Recursive function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);

        // Calling once
        fun(n - 1);

        // Calling twice
        fun(n - 1);
    }
}

// Driver code
int main()
{
    fun(3);
    return 0;
}
```

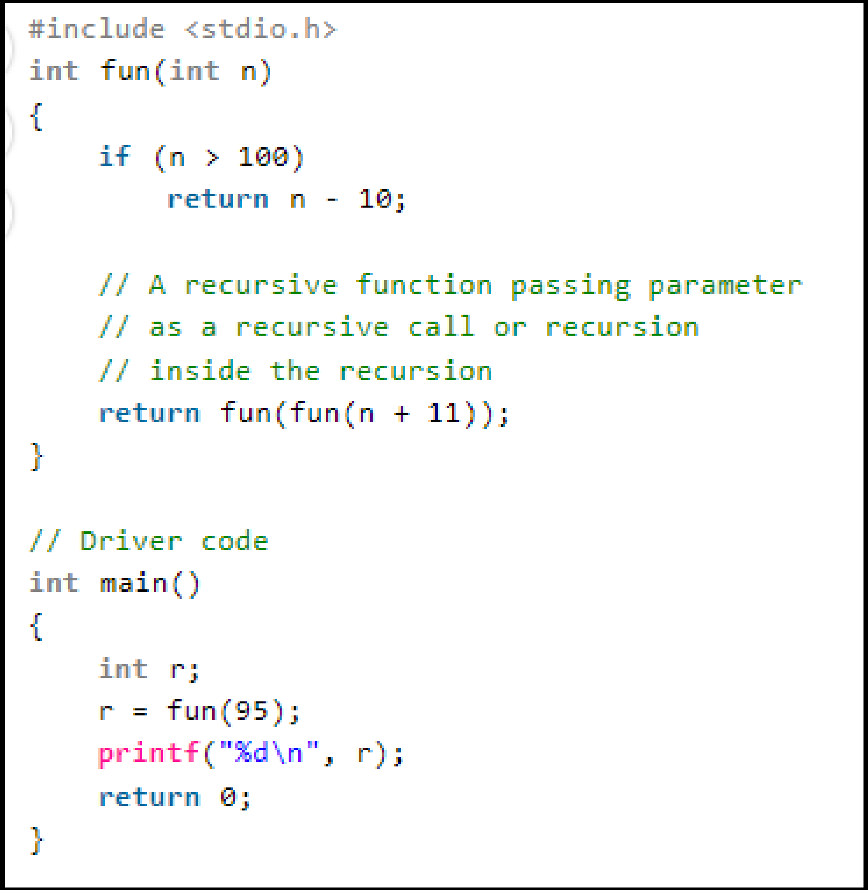
https://stackoverflow.com/questions/33923/what-is-tail-recursion

Tree Recursion

- Ex Workout Tracing Tree of Tree Recursion
- Refer
  - https://dotnettutorials.net/lesson/tree-recursion-in-c/

Nested Recursion

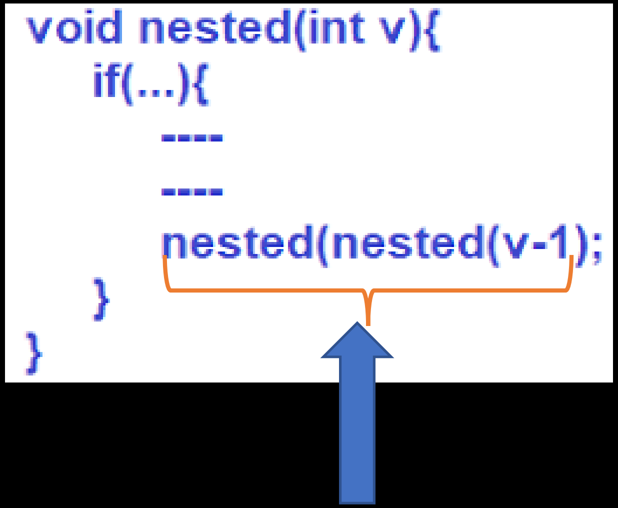
- **Nested Recursion**: In this recursion, a recursive function will pass the parameter as a recursive call. That means “recursion inside recursion”. Let see the example to understand this recursion.



```
#include <stdio.h>
int fun(int n)
{
    if (n > 100)
        return n - 10;

    // A recursive function passing parameter
    // as a recursive call or recursion
    // inside the recursion
    return fun(fun(n + 11));
}

// Driver code
int main()
{
    int r;
    r = fun(95);
    printf("%d\n", r);
    return 0;
}
```



```
void nested(int v){
    if(...){
        ----
        ----
        nested(nested(v-1);
    }
}
```

https://stackoverflow.com/questions/33923/what-is-tail-recursion

Applications of Recursion

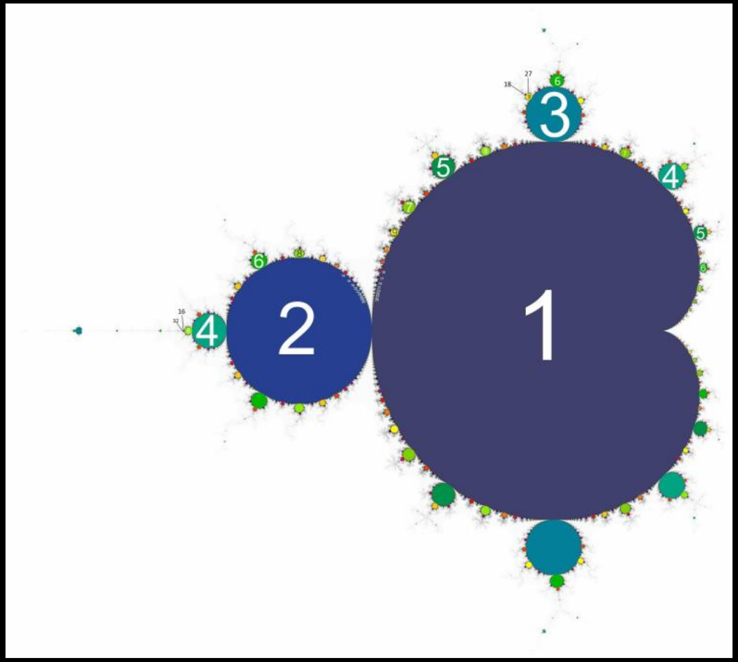
- Recursion is used in many fields of computer science and mathematics, which includes:
  1. Searching and sorting algorithms: Recursive algorithms are used to search and sort data structures like trees and graphs.
  2. Mathematical calculations: Recursive algorithms are used to solve problems such as factorial, Fibonacci sequence, etc.
  3. Compiler design: Recursion is used in the design of compilers to parse and analyze programming languages.
  4. Graphics: many computer graphics algorithms, such as fractals and the Mandelbrot set, use recursion to generate complex patterns.
  5. Artificial intelligence: recursive neural networks are used in natural language processing, computer vision, and other AI applications.

https://www.geeksforgeeks.org/what-is-recursion/

Refer: How do fractals play a role in the creation of realistic 3D models for computer graphics?
https://www.quora.com/How-do-fractals-play-a-role-in-the-creation-of-realistic-3D-models-for-computer-graph

Mandelbrot set

- The Mandelbrot set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules.



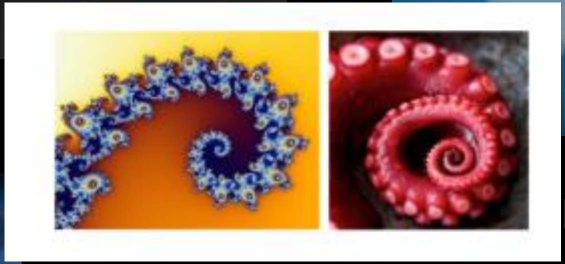
Refer : https://math.hws.edu/eck/js/mandelbrot/MB.html

https://math.hws.edu/eck/js/mandelbrot/MB-info.html

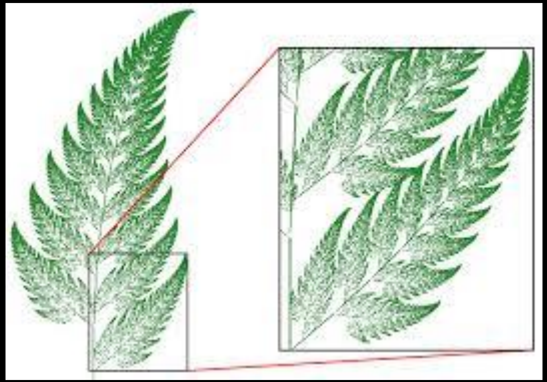
- **Fractal-generating software**
- https://en.wikipedia.org/wiki/Fractal-generating_software

- Fractal generating software creates mathematical beauty through visualization.
- Modern computers may take seconds or minutes to complete a single high resolution fractal image.
- Images are generated for both simulation (modeling) and random fractals for art. Fractal generation used for modeling is part of realism in computer graphics.
- Fractal generation software can be used to mimic natural landscapes with fractal landscapes and scenery generation programs.
- Fractal imagery can be used to introduce irregularity to an otherwise sterile computer generated environment.

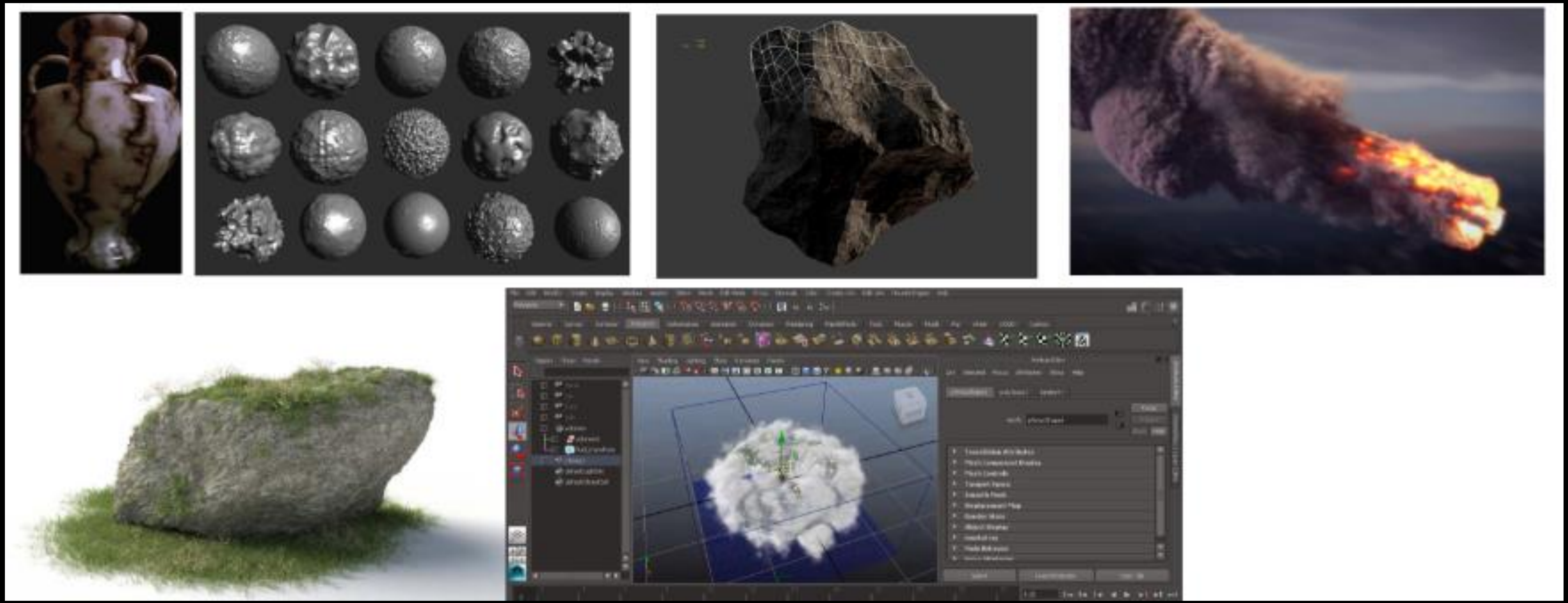
Fractal generated Graphics in 3D Computer Games?

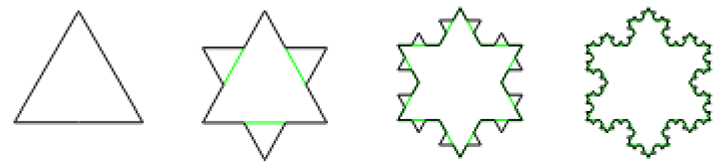


Ferns and Cactus Plants?=> self recursion

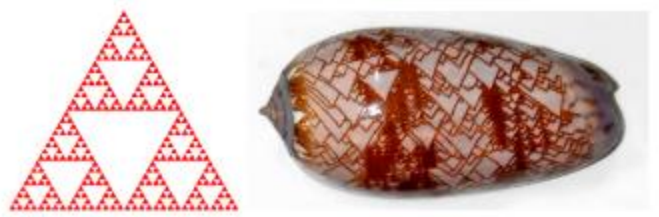


https://remptongames.medium.com/links-awakening-and-fractal-game-design-e3af668e6be9

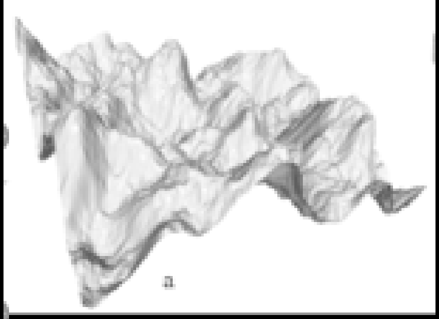




Koch Snowflake



Sierpinski triangle, Shell:Oliva Porphyria



Advantages of Recursion

- Understanding types of recursion can help in designing efficient and elegant recursive algorithms while considering the potential trade-offs
- Recursion can simplify complex problems by breaking them down into smaller, more manageable pieces.
- Recursive code can be more readable and easier to understand than iterative code.
- Recursion is essential for some algorithms and data structures.
- Also with recursion, we can reduce the length of code.

Disadvantages of Recursion

- Recursion can be less efficient than iterative solutions in terms of memory and performance.
- Recursive functions can be more challenging to debug and understand than iterative solutions.
- Recursion can lead to stack overflow errors if the recursion depth is too high.

https://www.geeksforgeeks.org/what-is-recursion/

Recursion – Some Best Practices

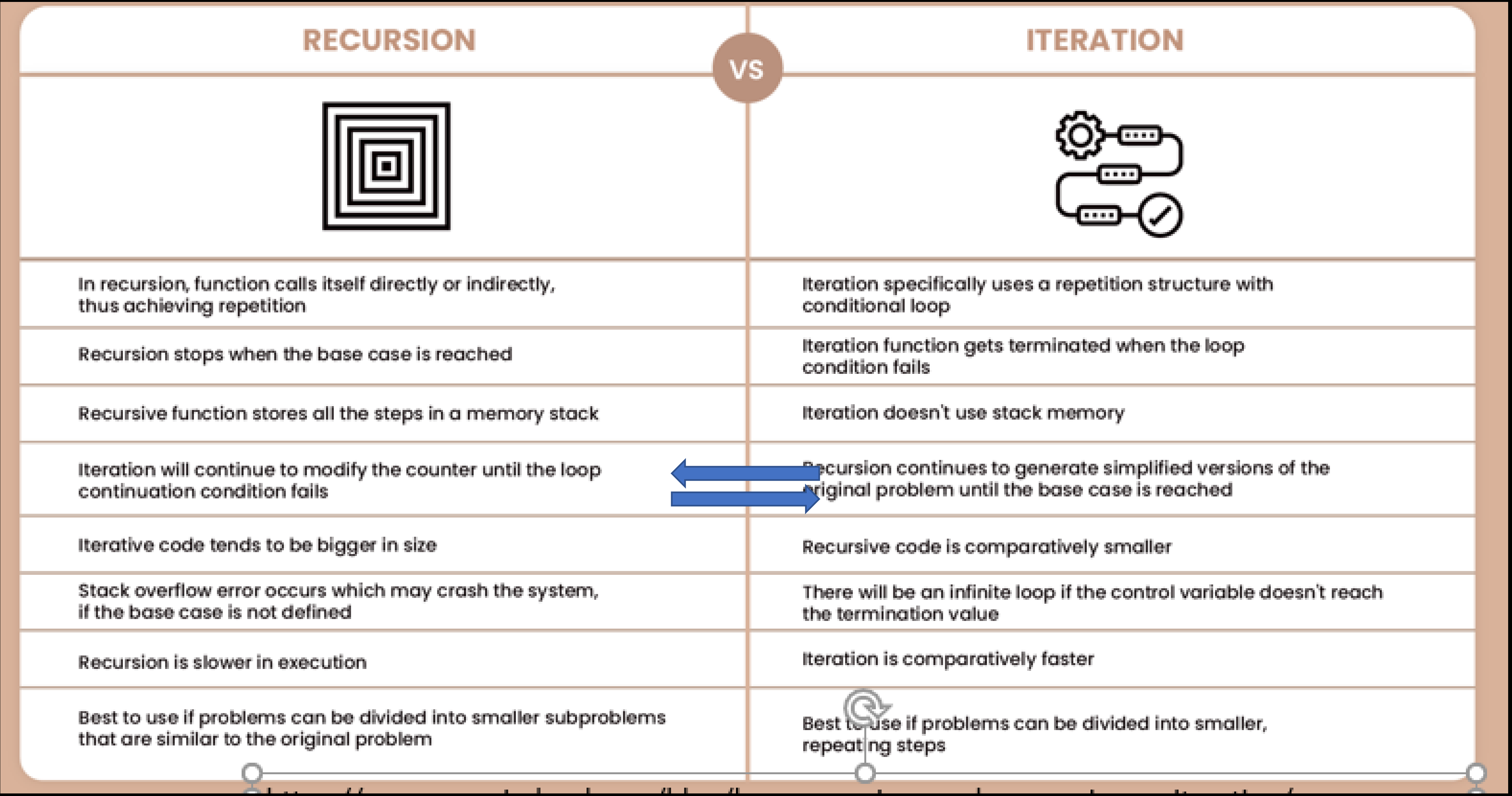
- Identify the Base Case(s) Carefully: Base case(s) provide the termination condition for recursion. Ensure that the base case condition is well-defined and reachable to avoid infinite recursion.
- Ensure Progress Towards the Base Case: In the recursive case(s), ensure that the problem is being divided into smaller subproblems that lead to reaching the base case eventually. Each recursive call should make progress toward the base case.
- Properly Manage Memory and Resources: Recursion may consume a significant amount of memory, especially if the recursive calls are nested deeply. Be mindful of memory usage and consider optimization techniques like tail recursion or memorization when applicable. (Memonisation covered in later lecture)
  - **What is memoization? A Complete tutorial**
  - https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/
- Test with Different Input Sizes: Recursion may have different performance characteristics depending on the input size. Test your recursive function with various input sizes to identify any potential performance bottlenecks.

https://intellipaat.com/blog/recursion-in-data-structure/

Recursion – Some Common Pitfalls

- Stack Overflow: If the recursion depth becomes too large, it can result in a stack overflow error. This happens when the call stack, which keeps track of function calls, exceeds its memory limit. Ensure that your recursive function terminates within reasonable recursion depths to avoid this issue.
- Redundant or Incorrect Recursive Calls: Be careful with the recursive function calls within the function body. Make sure the arguments passed to the recursive call are appropriate and lead to a valid progression toward the base case. Incorrect or redundant recursive calls can lead to incorrect results or infinite recursion.
- The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.

https://intellipaat.com/blog/recursion-in-data-structure/

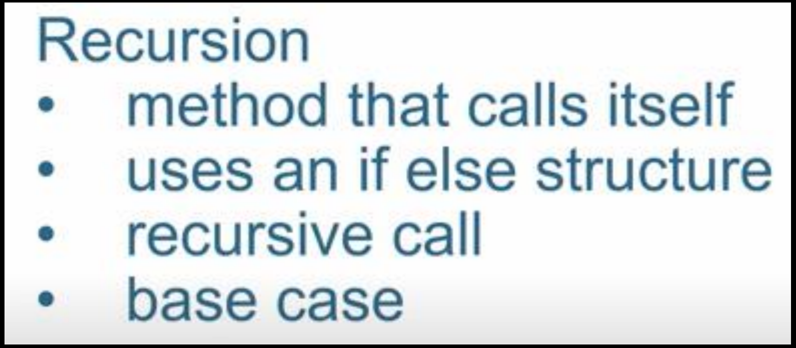


RECURSION	ITERATION
In recursion, function calls itself directly or indirectly, thus achieving repetition	Iteration specifically uses a repetition structure with conditional loop
Recursion stops when the base case is reached	Iteration function gets terminated when the loop condition fails
Recursive function stores all the steps in a memory stack	Iteration doesn't use stack memory
Iteration will continue to modify the counter until the loop continuation condition fails	Recursion continues to generate simplified versions of the original problem until the base case is reached
Iterative code tends to be bigger in size	Recursive code is comparatively smaller
Stack overflow error occurs which may crash the system, if the base case is not defined	There will be an infinite loop if the control variable doesn't reach the termination value
Recursion is slower in execution	Iteration is comparatively faster
Best to use if problems can be divided into smaller subproblems that are similar to the original problem	Best to use if problems can be divided into smaller, repeating steps

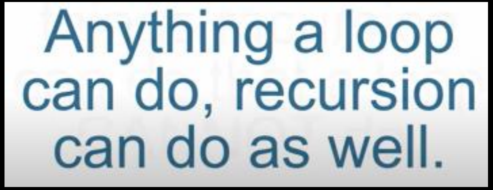
https://stackoverflow.com/questions/33923/what-is-tail-recursion

Recursion vs Loops/Iteration

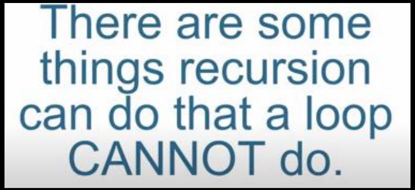
Loops
- Loop control variable
- Other Boolean event
- while
- do while
- for



Recursion
- method that calls itself
- uses an if else structure
- recursive call
- base case



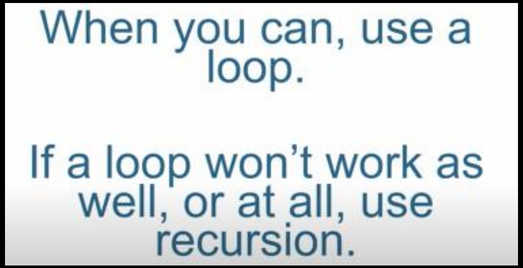
Anything a loop can do, recursion can do as well.



There are some things recursion can do that a loop CANNOT do.

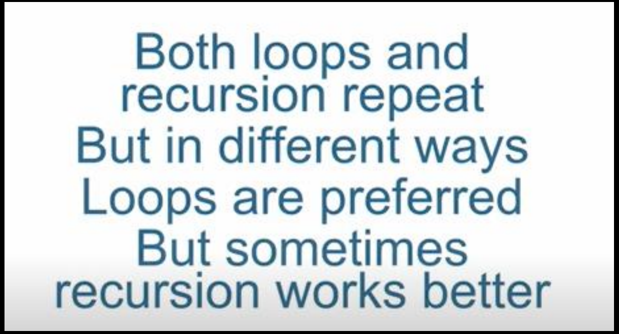
A loop is memory efficient.

Recursion is memory costly.



When you can, use a loop.

If a loop won't work as well, or at all, use recursion.



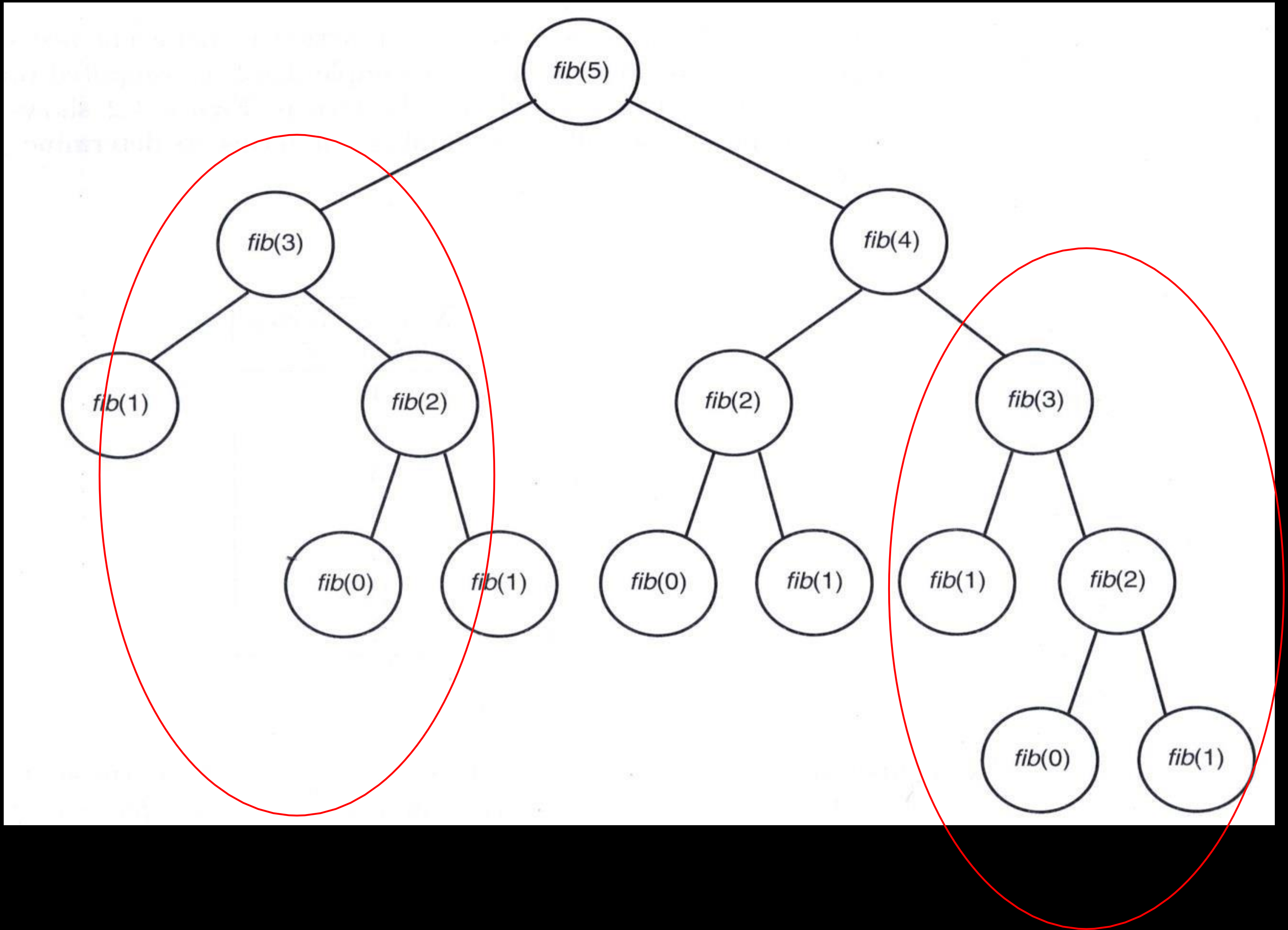
Both loops and recursion repeat
But in different ways
Loops are preferred
But sometimes recursion works better

1st Solution to the n-th Fib. Number (Recursive version)

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

Recursion in 100 Seconds
https://www.youtube.com/watch?v=rf60MejMz3E

The Recursion Tree when Computing the Fifth Fibonacci Term.



2nd Solution to the n-th Fib. Number (Iterative version)

```
int fib2 (int n) {
  index i;
  int f[0..n];
  f[0] = 0;
  if (n > 0) {
    f[1] = 1;
    for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Why Analyze an Algorithm?

- The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application.
- Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements.
- Algorithms tend to become shorter, simpler, and more elegant during the analysis process

Analysis of Algorithms

- Time complexity analysis
  - Determination of how many times the basic operation is done for each value of the input size.
  - Should be independent of CPU, OS, Programming languages…

- Metrics
  1. Basic operation
     - Comparisons, assignments, etc.
  2. Input size

     e.g
     - The number of elements in an array
     - The length of a list
     - The number of columns and rows in a matrix
     - The number of vertices and edges in a graph

Analysis of Algorithms ctd..

1. Worst-case analysis
2. Average-case analysis
3. Best-case analysis

Worst Case Analysis **(Pessimistic Scenario)** - Mostly used

- This is the exact opposite of the best case. It considers the most challenging or demanding situation for the algorithm.
- The worst case analysis provides an upper bound on the time or resources needed for the algorithm's completion which is good information.
- Example :
  - A worst-case scenario for a sorting algorithm would be sorting a list that is in completely reverse order.
  - For Linear Search, the worst case happens when the element to be searched (x) is the last element or not present in the array. When x is not present, the search() function compares it with all the elements of array one by one. Therefore, the worst-case time complexity of the linear search would be O(n).
- We must know the case that causes a maximum number of operations to be executed.

Significance of Worst Case Scenarios

- Worst case analysis is a fundamental aspect of evaluating algorithm performance.
- It involves determining how an algorithm behaves under the most adverse conditions.
- This type of analysis examines the maximum number of operations an algorithm might perform or the maximum amount of resources it might require.
- It answers a critical question: "In the least favorable scenario, how badly could this algorithm perform?"
- This is crucial for applications where predictability and reliability are very important, such as in mission critical applications/life critical medical applications or important financial applications.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Worst Case Examples from Widely-Used Algorithms

- 1. Linear Search: In a linear search algorithm, the worst case occurs when the element being searched for is not present in the array or is at the very end. The algorithm has to check each element, leading to a time complexity of $O(n)$.
- 2. Sorting Algorithms (like Quick Sort): For Quick Sort, the worst case happens when the pivot element selected at each step is the smallest or largest element, leading to imbalanced partitions. This results in a time complexity of $O(n^2)$. (Detailed under sorting example later lecture)
- 3. Graph Algorithms : In graph search algorithm for finding the shortest path, the worst case is when the algorithm has to traverse all vertices and edges in the graph, leading to a time complexity of $O(V^2)$ for basic implementations, where V is the number of vertices. (you will lean algorithms like Dijkstra's Algorithm in later lessons)

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Importance of Worst Case Analysis in Algorithm Design

- Worst case analysis is often considered the most crucial aspect of algorithm design for several reasons:
  - 1. Reliability and Robustness: It ensures that an algorithm's performance is predictable even in the most challenging scenarios, which is critical for maintaining system stability and reliability.
  - 2. Resource Allocation: Understanding the worst case helps in adequately provisioning resources such as memory and processing power, ensuring that the system can handle the most resource-intensive scenarios.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Importance of Worst Case Analysis in Algorithm Design

- 3. Risk Management: In critical applications, worst case scenarios help identify potential risks and bottlenecks, allowing developers to mitigate these risks early in the design phase.
- 4. Performance Guarantees: Worst case analysis offers a conservative estimate of algorithm performance, providing a safety net for both developers and users. It is particularly useful when the input data characteristics are <u>unknown</u> or highly variable.
- 5. Comparison and Selection of Algorithms: In many cases, especially in competitive programming or system optimization, the choice of an algorithm is based on its worst case performance, as it provides a clear metric for comparison.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Importance of Worst Case Analysis in Algorithm Design

- Worst case analysis is a vital tool in algorithm evaluation, providing essential insights into the limits of an algorithm's performance.
- By considering the most challenging scenarios, developers can design algorithms that are not only efficient under typical conditions but also robust and reliable under extreme conditions.
- This comprehensive understanding is key to creating high-performance, resilient systems and applications.

Average Case Analysis (Realistic Scenario)

- Falling between the best and worst extremes, the average case analysis attempts to ascertain the algorithm's performance under 'typical' or 'average' conditions.
- It's often more complex to compute as it requires understanding of how the inputs are distributed.
  - E.g The average case for the linear search would involve calculating the expected number of comparisons for a randomly chosen target in the array.
- The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

Average Case Analysis (Realistic Scenario)

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.
- Sum all the calculated values and divide the sum by the total number of inputs.
- We must know (or predict) the distribution of cases.
- For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1).

Limitations in Applicability of Average Case

- The usefulness of average case analysis can be limited if the actual input distribution significantly differs from the assumed distribution, or if the distribution of inputs is unknown.
- Average case analysis plays a crucial role in understanding and predicting the practical performance of algorithm .
- While it may be complex to derive, its insights are invaluable in guiding the development of efficient and realistic algorithms suited for the typical conditions they will encounter in actual use.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Best Case Analysis (Optimistic Scenario) - Very Rarely used

- This refers to the scenario where the algorithm performs its task under the most favorable conditions.
- It represents the minimum time or resources required for the algorithm to complete its execution.
  - For example, in the best case, a search algorithm might find the target element in the first place it looks.
- In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
- In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be 1
- Guaranteeing a lower bound on an algorithm does not provide any information as in the worst case, an algorithm may take years to run.

Examples of Best Case Analysis in Common Algorithms

- 1. **Linear Search**: The best case in a linear search algorithm occurs when the target element is the first element in the list. Here, the algorithm only makes one comparison, regardless of the list's size.
- 2. **Binary Search**: For binary search, the best case happens when the middle element (first checked) is the target element. Even for large arrays, the search completes after just one comparison.
- 3. **Sorting Algorithms (like Bubble Sort)**: In bubble sort, the best case occurs when the array is already sorted. The algorithm goes through the list once without making any swaps, thus completing in linear time.

Why Best Case Analysis Can Be Misleading in Practical Applications

- While best case analysis provides an optimistic view of an algorithm's potential efficiency, it can be misleading for several reasons:
  - 1. Unrealistic Expectations: The best case scenario often represents a highly idealized and rare situation. Relying solely on this analysis might create unrealistic expectations about the algorithm's typical performance.
  - 2. Not Reflective of Average Performance: In most real-world applications, inputs to algorithms are not systematically arranged to meet the best case criteria. Thus, the best case does not accurately reflect the average or expected performance of the algorithm.
  - 3. Overemphasis on Optimistic Outcomes: Focusing too much on best case scenarios can lead to neglecting more common and challenging cases, resulting in a lack of preparation for the algorithm's actual operational environment.
  - 4. Resource Allocation and Planning: In system design and resource allocation, planning based on best case scenarios can result in inadequate resource provisioning, potentially leading to performance issues in average or worst-case scenarios.
- While best case analysis is a part of a comprehensive algorithm evaluation, it should be considered alongside worst and average case analyses to gain a realistic understanding of an algorithm's performance.
- It serves more as a theoretical benchmark rather than a sole indicator of practical efficiency, and its limitations must be recognized in the broader context of algorithm design and application.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Comparative Analysis of Best, Worst, and Average Cases

- Understanding the differences and relative importance of best, worst, and average case analyses is crucial for a comprehensive evaluation of algorithms. A side-by-side comparison highlights their distinct roles and how they collectively contribute to informed decision-making in algorithm selection and design.

1. **Best Case Analysis**:
- **Focus**: Optimistic scenario where the algorithm performs with the least effort.

- **Usage**: Often used for theoretical benchmarks and understanding the lower limit of an algorithm's efficiency.

- **Limitation**: Can be misleading as it does not reflect typical or challenging scenarios.

2. Worst Case Analysis:
- Focus: Pessimistic scenario considering the most demanding input.

- Usage: Crucial for ensuring reliability and performance guarantees, especially in critical applications.

- Advantage: Provides an upper bound on resource requirements, ensuring that the algorithm can handle the toughest scenarios.

3. Average Case Analysis:
- Focus: Realistic scenario, taking into account the probability distribution of all possible inputs.

- Usage: Offers a balanced view of algorithm performance in typical use.

- Complexity: Requires knowledge of or assumptions about input distribution, making it more complex to calculate.

Refer: https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Time Complexity Analysis: (In Big-O notation)
Linear Search

- Best Case: This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.
- Average Case: This will take place if the element to be searched is on the middle index of the given list.
- Worst Case: This will take place if:
  - The element to be searched is on the last index
  - The element to b e searched is not present on the list

Importance of Analyses in Algorithm Evaluation

- Understanding these three facets of algorithm performance is essential for a number of reasons:
  1. Comprehensive Performance Assessment: Together, these analyses offer a complete picture of an algorithm’s efficiency. While the worst case analysis provides a guarantee of sorts about the algorithm's limits, the best and average cases offer a more complex view of its everyday performance.
  2. Informed Algorithm Selection: In software development, choosing the right algorithm for a given task is crucial. These analyses help in selecting the most appropriate algorithm based on its expected performance in various scenarios.
  3. Resource Allocation: Especially in systems with limited resources, knowing the worst case resource needs is vital. It helps in ensuring that the system can handle the most demanding scenarios.
  4. Realistic Expectations: While worst case analysis is critical, it can sometimes be overly pessimistic. Average case analysis, on the other hand, provides a more realistic expectation of how the algorithm will perform in everyday use.
  5. Algorithm Optimization: Understanding the different performance scenarios can guide developers in optimizing algorithms. For instance, if the worst case performance is significantly lagging, efforts can be directed towards mitigating these cases.
- Best, worst, and average case analyses are important tools in algorithm design and evaluation. They provide vital insights into the behavior of algorithms under various conditions, guiding developers in creating more efficient and effective solutions.

https://www.linkedin.com/pulse/dsa-mastery-deciphering-algorithm-performance-best-worst-manish-v--v51bf

Popular Notations in Complexity Analysis of Algorithms

- 1. Big-O Notation
- We define an algorithm’s worst-case time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.
- 2. Omega Notation
- It defines the best case of an algorithm’s time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.
- 3. Theta Notation
- It defines the average case of an algorithm’s time complexity, the Theta notation defines when the set of functions lies in both O(expression) and Omega(expression), then Theta notation is used. This is how we define a time complexity average case for an algorithm.

https://www.geeksforgeeks.org/worst-average-and-best-case-analysis-of-algorithms/

Worst-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, n
  - Worst case happens if *x* is in the last element of *S[n]*
  - Therefore, $W(n) = n$.

Average-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, n
  - Average case
    - Assume each item in the array is distinctive
    - Case 1: when x is always in *S[n]*
      - The probability that *x* is the *k*-th element in *S[n]* $= {}^{1}\!/_{n}$
      - The number of operations if *x* is the k^{th} = *k* times
      - Therefore,

$$A(n) = \sum_{k=1}^{n} k \times \frac{1}{n} = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Average-case analysis (Cont'd)

- Case 2: *x* is either in *S[n]* or not in *S[n]*
  - The probability that *x* is in *S[n]*: *p*
    - The probability that *x* is in the *k*-th position of *S[n]* = *p/n*
      - The probability that x is not in *S[n]* = *1 – p*
  - Therefore , $A(n) = \sum_{k=1}^{n} (k \times \frac{p}{n}) + n(1-p)$

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p)$$

$$= n(1-\frac{p}{2}) + \frac{p}{2}$$

  - $p = 1 \Rightarrow A(n) = (n+1)/2$
    $p = 1/2 \Rightarrow A(n) = 3n/4 + 1/4$

Best-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, `n`
  - Best case happens if *x* is the first element of *S[n]*, i.e. *S[1]*
  - Therefore, *B(n) = 1*.

Review: time complexity analysis

- Among possible analysis, WCA, ACA, and BCA, which one is the right one?
- Think about …
  - you are working for a nuclear power plant.
  - what if you are working for an ecommerce portal
- Which one do you think is the most useful?
- Which one do you think is the hardest to analyze?

Analysis of Correctness

- Efficiency analysis vs. Correctness analysis
- In this course, when it is referred to algorithm analysis, it refers to efficiency analysis
- We can also analyze the correctness of an algorithm by developing a mathematical proof that the algorithm actually does what it is supposed to do.
- An algorithm is *incorrect*…
  - if it does not stop for a given input or
  - if it gives a wrong answer for an input.

Questions?