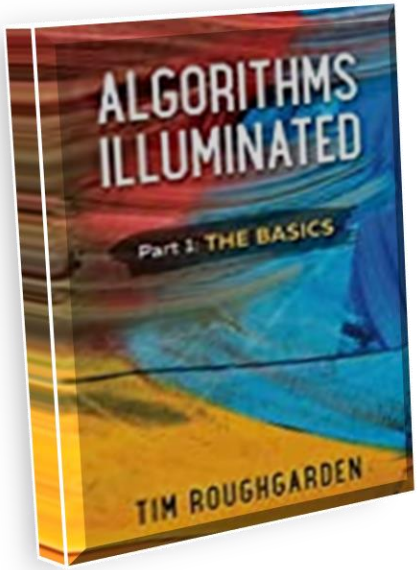
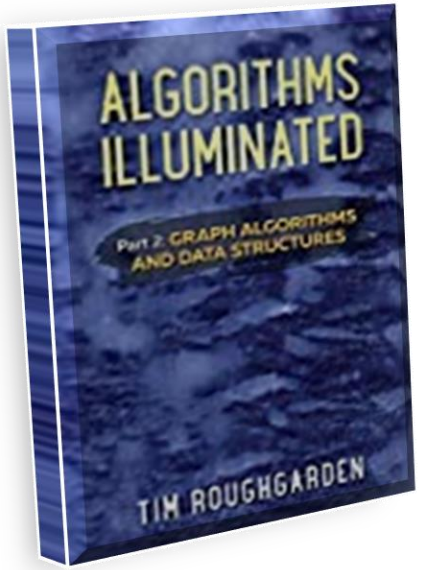
Insertion Sort-Dance

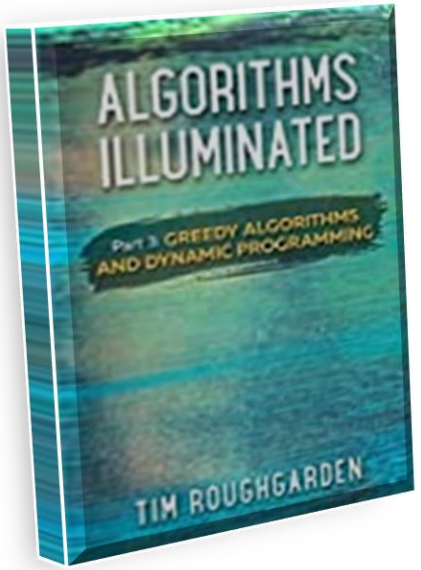
- https://www.youtube.com/watch?v=ROalU379l3U

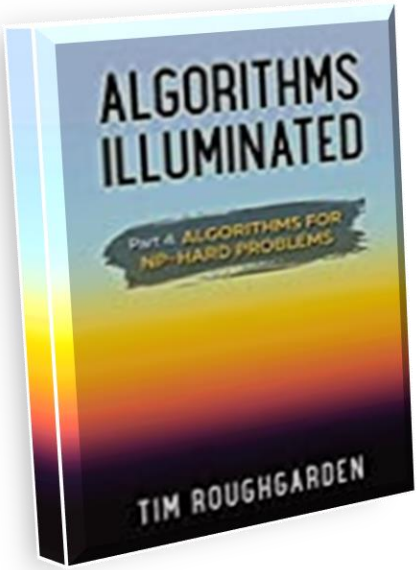
Textbooks

- Algorithms Illuminated , Tim Roughgarden:
  - Part 1: The Basics (September 2017).
  - Part 2: Graph Algorithms and Data Structures (August 2018)
  - Part 3: Greedy Algorithms and Dynamic Programming (May 2019)
  - Part 4: Algorithms for NP-Hard Problems (July 16, 2020)
  - http://algorithmsilluminated.org/









Reference

Introduction to Algorithms

3rd edition

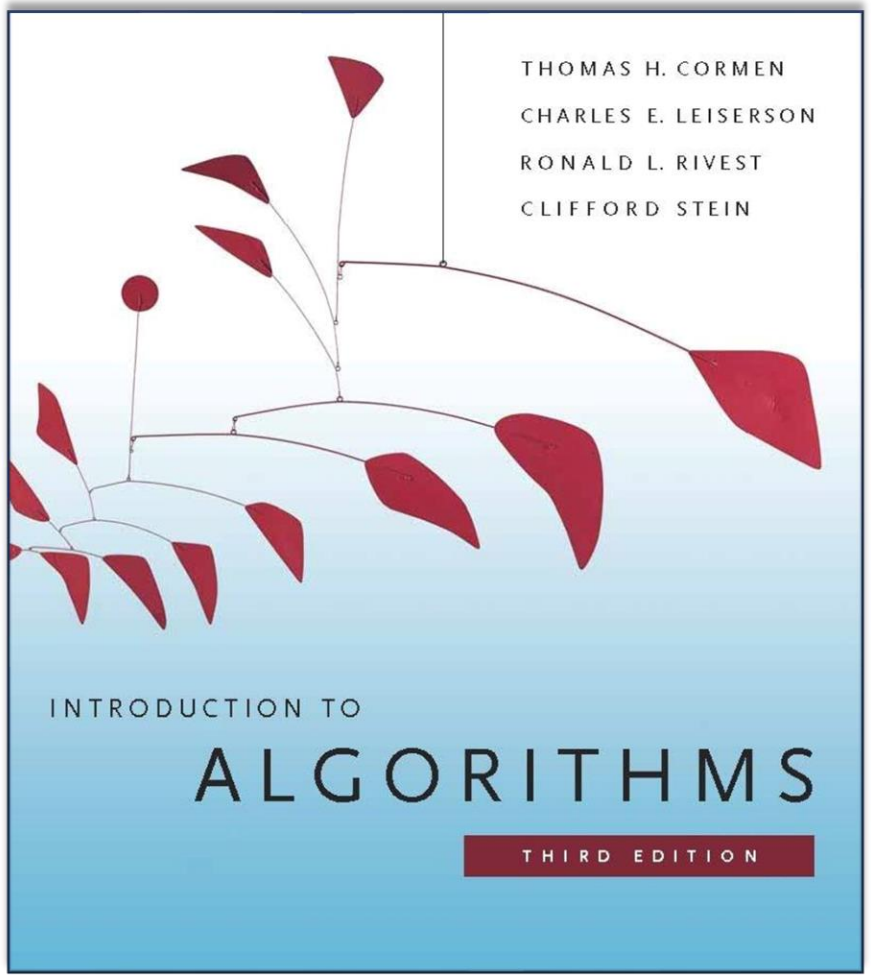
By: Thomas H. Cormen,

Charles E. Leiserson,

Ronald L. Rivest,

Clifford Stein

Published by The MIT Press, 2009



Recommended Video Lectures

1) MIT

"Introduction to Algorithms"

Prof. Erik Demaine,Prof. Srini Devadas ,2011

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

2) MIT

"Introduction to Algorithms"

Prof. Charles Leiserson

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/

3) Stanford-Coursera

"Algorithm Specialization"

Tim Roughgarden

https://www.coursera.org/specializations/algorithms

Recommended Video Lectures

4) UC San Diego and National Research University.

"Data Structures and Algorithms Specialization"

https://www.edx.org/micromasters/ucsandiegox-algorithms-and-data-structures

https://www.coursera.org/specializations/data-structures-algorithms

5) Ghassan Shobaki Computer Science Lectures

https://www.youtube.com/playlist?list=PL6KMWPQP_DM8t5pQmuLlarpmVc47DVXWd

References

- CS 161, Mary Wootters, Design and Analysis of Algorithms, Stanford University, Fall 2017.

http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/

Contact Info.

- E-mail

  ShahiraAzazy@gmail.com

Introduction

Introduction

- Factors that affects program performance:
  - Hardware
  - Compilers
  - Programming Language
  - Operating Systems
  - Data structures
  - **Algorithms**

Definition

An algorithm is any well-defined computational procedure that takes Some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output

History

- The scientist, astronomer and mathematician Muḥammad ibn Musa Al-Khwarizmi ““father of algebra” ” , formerly Latinized as Algorithmi, under the patronage of the Caliph Al-Ma'mun of the Abbasid Caliphate. Around 820 AD he was appointed as the astronomer and head of the library of the House of Wisdom in Baghdad. Al-Khwarizmi's in his book, “The Compendious Book on Calculation by Completion and Balancing”, presented the first systematic solution of linear and quadratic equations.
- He wrote several influential books “On the Calculation with Hindu Numerals”, which described how to do arithmetic using Arabic numerals. The original manuscript was lost, though a Latin translation from the 1100’s introduced this number system to Europe. The old French word algorisme meant “the Arabic numerals system”, and only later did it come to mean a general recipe for solving computational problems.

Why we study algorithm?

In a December 2010 report to the United States White House, the President's council of advisers on science and technology wrote the following:

"Everyone knows Moore's Law — a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years. . . in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed."[1]

1) Excerpt from Report to the President and Congress: Designing a Digital Future, December 2010 (page 71).

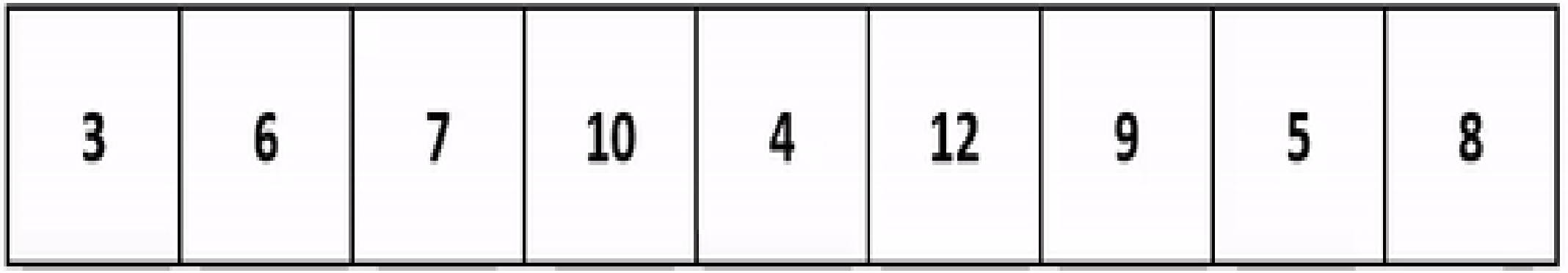
Motivation

Motivation

- If you want to be a world-class programmer, you can program every day for ten years, or you can program every day for two years and take an algorithms class.

Motivation

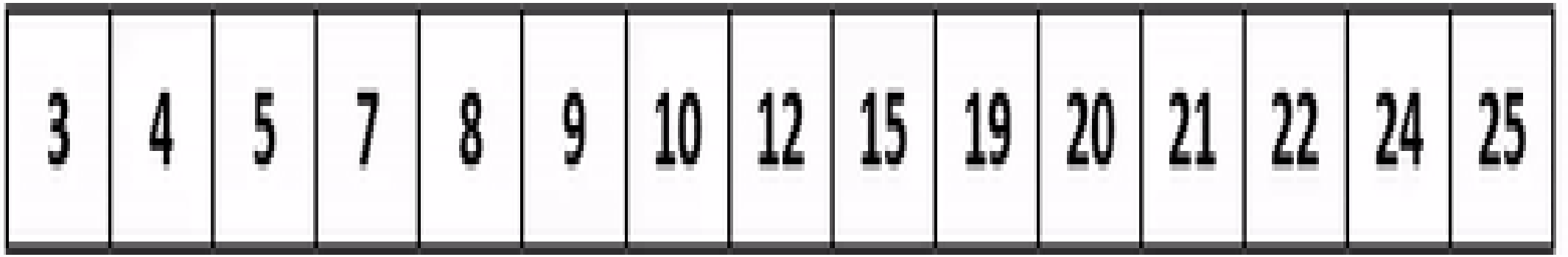
Linear Search



1.) let's find 5 with linear search algorithm

https://ramnauth.github.io/cs%20101/2018/10/17/sorting-searching/

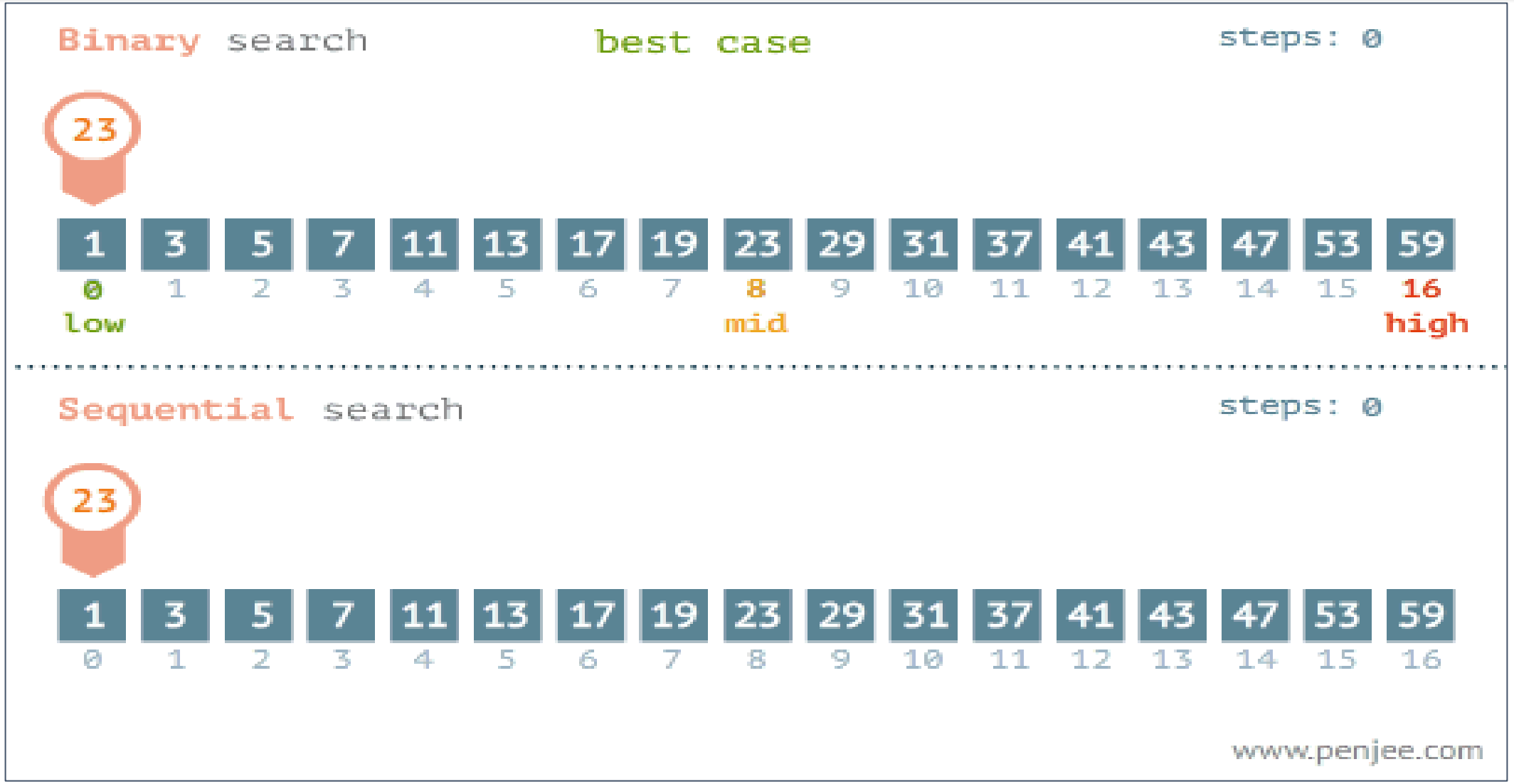
Binary Search



2.) let's find 22 with binary search algorithm

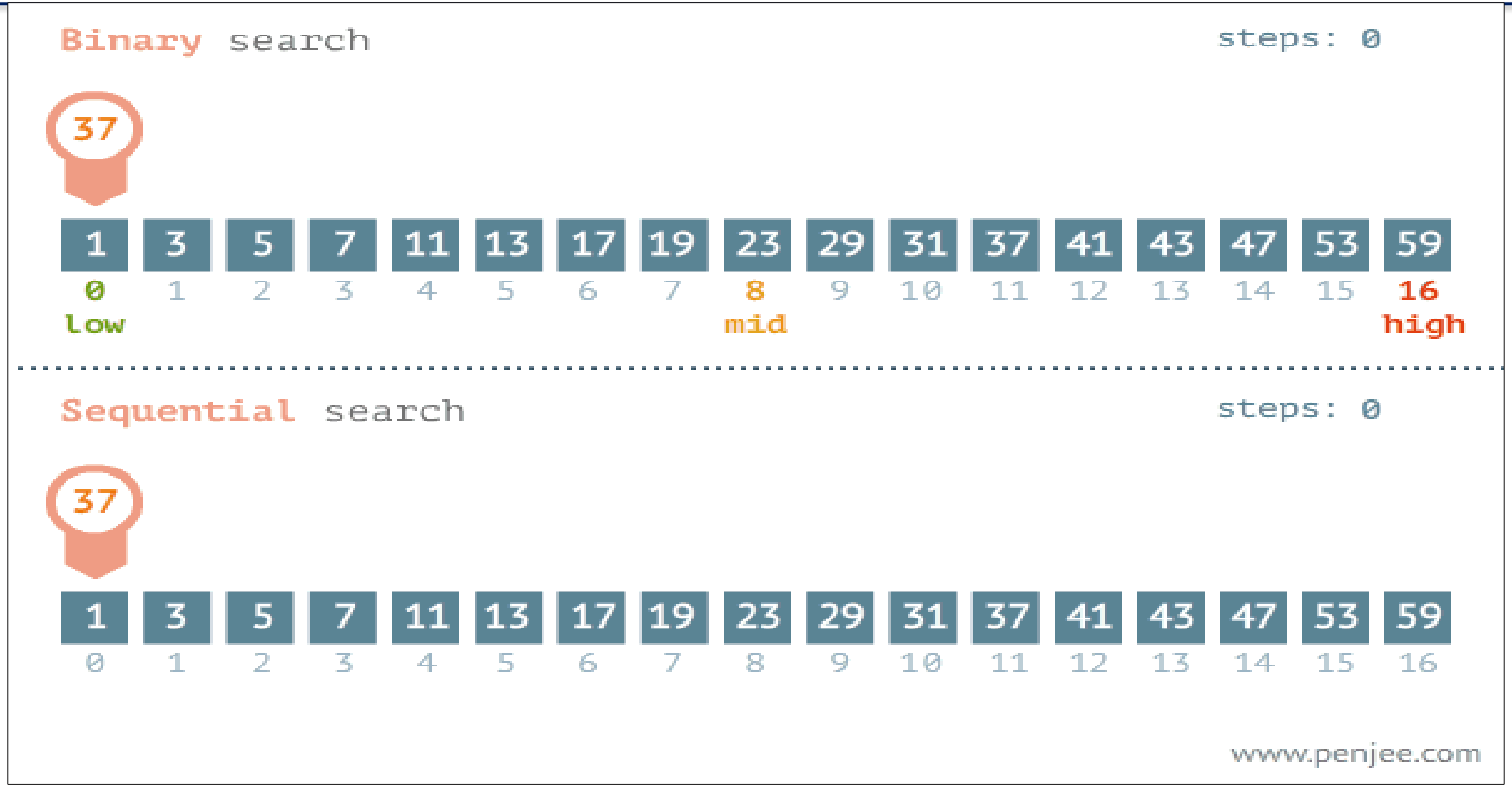
- https://ramnauth.github.io/cs%20101/2018/10/17/sorting-searching/

Linear vs Binary Search (best case)



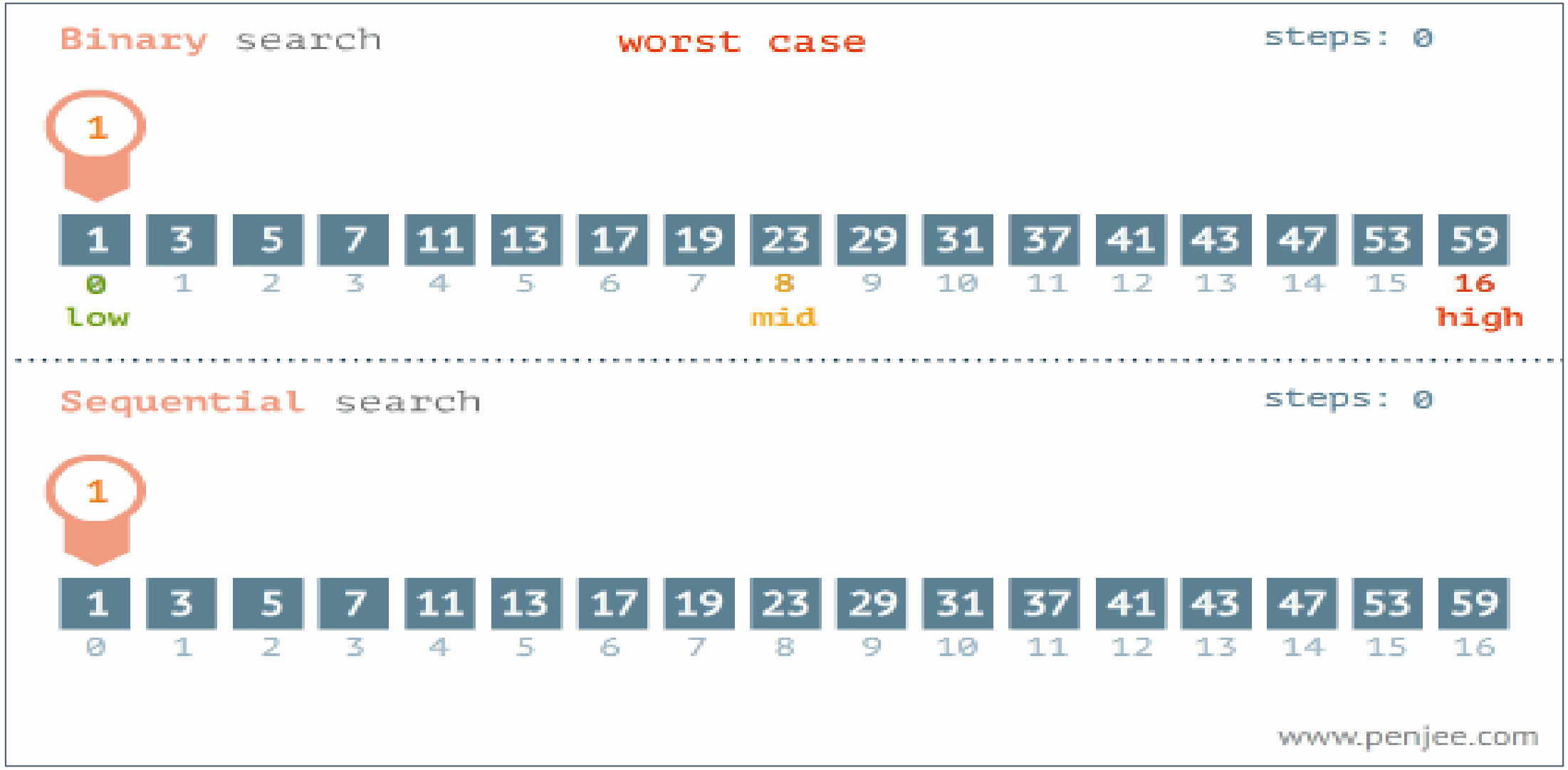
- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

Linear vs Binary Search (average case)



- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

Linear vs Binary Search (worst case)



- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

Linear vs Binary Search

- Choose One:
  - Linear Search
  - Binary Search(Sort the array)

WHAT DOES A COMPUTER DO

- Fundamentally:
  - Performs calculations
    - a billion calculations per second!
    - two operations in same time light travels 1 foot
  - Remembers results
    100s of gigabytes of storage!
    typical machine could hold 1.5M books of standard size

Edx-MIT: Introduction_to_Computer_Science_and_Programming_Using_Python

https://ohthehumanityblog.files.wordpress.com/2014/09/computerthink.gif

SIMPLE CALCULATIONS ENOUGH?

- Searching the World Wide Web
  - 45B pages; 100 words/page; 10 operations/word to find
  - Need 5.2 days to find something using simple operations
- Playing chess
  - Average of 35 moves/setting; look ahead 6 moves; 1.8B boards to check; 100 operations/choice
  - 30 minutes to decide each move
- Good algorithm design also needed to accomplish a task!

ENOUGH STORAGE?

What if we could just pre-compute information and then look up the answer

- Playing chess as an example
- Experts suggest 10^123 different possible games
- Only 10^80 atoms in the observable universe

Algorithmic toolkit…

- Efficient procedures for solving problems on large inputs (Ex: U.S. Highway Map,Human Genome)

Limitation of Asymptotic Notation

Assumptions:

1. All operation takes the same amount of time.
2. Ignore constants and lower order terms.

Definition

- $f_n = \begin{cases} o & n = 0 \\ 1 & n = 1 \\ f_{n-1} + f_{n-2}, & n > 1 \end{cases}$

- Fibonacci series ➔0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
- Developed to Study Rabbit Populations

- Note that there is a slightly different definition of Fibonacci numbers: there, *F*0=*F*1=1.

Computing Fibonacci numbers

- Input: An integer n ≥ 0.
- Output: Fn.
- Algorithm:

```
FibRecurs(n)
-------------
if n ≤ 1:
    return n
else:
    return FibRecurs(n − 1) + FibRecurs(n − 2)
```

$$T(n)=\begin{cases} 2 & \text{if } n \le 1 \\ T(n-1)+T(n-2)+3 & else \end{cases}$$

- Running time : Let T(n) denote the number of lines of code executed by FibRecurs(n)

Example

- F20 = 6765
- F50 = 12586269025
- F100 = 354224848179261915075

- F500=139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125

- F999=2686381002448535938614672720214292396761660931898695234012317599761798170024788168933836965448335656419182785616144335631297667364221035032463485041037768036733415117289916972319708276398561576445007847417462 6

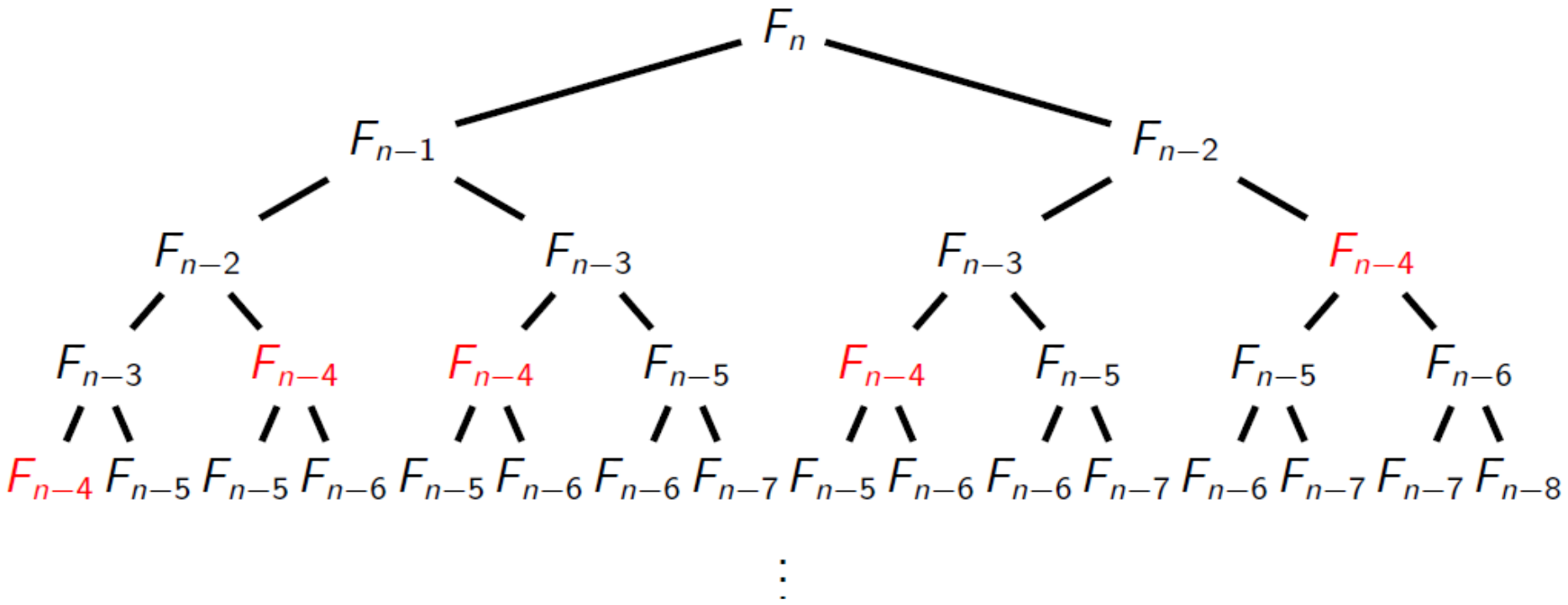
Why so slow?

- T(100) ≈ 1.77 $\mathbf{10^{21}}$(1.77 sextillion)
- Takes 56,000 years at 1GHz.

- Fibonacci Visualization
  - https://www.cs.usfca.edu/~galles/visualization/DPFib.html

https://www.coursera.org/specializations/data-structures-algorithms

Why so slow?


F_n
F_{n-1} F_{n-2}
F_{n-2} F_{n-3} F_{n-3} F_{n-4}
F_{n-3} F_{n-4} F_{n-4} F_{n-5} F_{n-4} F_{n-5} F_{n-5} F_{n-6}
F_{n-4} F_{n-5} F_{n-5} F_{n-6} F_{n-5} F_{n-6} F_{n-6} F_{n-7} F_{n-5} F_{n-6} F_{n-6} F_{n-7} F_{n-6} F_{n-7} F_{n-7} F_{n-8}
⋮

Efficient Algorithm

FibList(n)

create an array F [0 . . . n]

F [0] ← 0

F [1] ← 1

for i from 2 to n:

F [i] ← F [i − 1] + F [i − 2]

return F [n]

T(n) = 2n + 2. So T(100) = 202.

Efficiency of Programs

- Challenges in understanding efficiency of solution to a computational problem:
  - a program can be implemented in many different ways
  - you can solve a problem using only a handful of different algorithms
  - Different configuration of computers
  - Input

Input(1)

```
def linearSearch(List, x):
    for e in L:
        if e == x:
            return True
    return False
```

Best Case Run Time

1. **linearSearch([14, 15, 6, 27, 13, 16, 25, 11, 7], 15)**
2. **linearSearch([21, 1, 25, 22, 30, 13, 7, 24, 12], 24)**
3. **linearSearch([20, 10, 1, 7, 4, 22, 25, 12, 31], 20)**
4. **linearSearch([9, 3, 12, 24, 7, 8, 23, 11, 19], 8)**
5. **linearSearch([4, 12, 20, 17, 9, 14, 7, 24, 6], 7)**
6. **linearSearch([13, 9, 22, 3, 10, 17, 11, 2, 12], 26)**

Solution:3

Input(2)

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

2.Worst Case Run Time

1. **linearSearch([14, 15, 6, 27, 13, 16, 25, 11, 7], 15)**
2. **linearSearch([21, 1, 25, 22, 30, 13, 7, 24, 12], 24)**
3. **linearSearch([20, 10, 1, 7, 4, 22, 25, 12, 31], 20)**
4. **linearSearch([9, 3, 12, 24, 7, 8, 23, 11, 19], 8)**
5. **linearSearch([4, 12, 20, 17, 9, 14, 7, 24, 6], 7)**
6. **linearSearch([13, 9, 22, 3, 10, 17, 11, 2, 12], 26)**

Solution:6

Input(3)

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

3.Average Case Run Time

1. **linearSearch([14, 15, 6, 27, 13, 16, 25, 11, 7], 15)**
2. **linearSearch([21, 1, 25, 22, 30, 13, 7, 24, 12], 24)**
3. **linearSearch([20, 10, 1, 7, 4, 22, 25, 12, 31], 20)**
4. **linearSearch([9, 3, 12, 24, 7, 8, 23, 11, 19], 8)**
5. **linearSearch([4, 12, 20, 17, 9, 14, 7, 24, 6], 7)**
6. **linearSearch([13, 9, 22, 3, 10, 17, 11, 2, 12], 26)**

Solution:4

Input

- Which Input to Use to Evaluate ?
- Different inputs change how the program runs:
  - Search the first element in the list ==> BEST CASE
  - Search element not in list ==> WORST CASE
  - AVERAGE CASE????

How To Evaluate Efficiency of Programs

GOAL: to evaluate different algorithms

- Measure with a timer
- Count the operations
- Abstract notion of order of growth

Measure with a timer(1)

```
import time
t0 = time.clock()
sum=0
for i in range(10000000):
    sum+=i
t1 = time.clock() -t0
print("Time of Running :
",t1)
```

import time

- use time module
- importing means to bring in that class into your own file

t0 = time.clock()

- Startclock

for i in range(10000000):

- Loop

Measure with a timer(2)

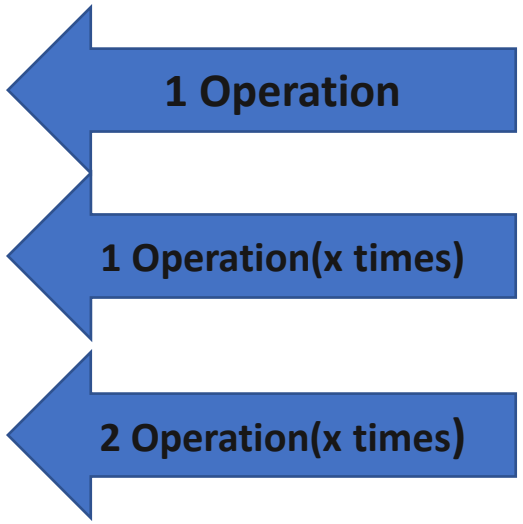
- Running time varies implementation (Programming Languages)
- Running time varies between computers
- Time varies for different inputs but cannot really express a relationship between inputs and time

Counting Operations(1)

- Assume these steps take constant time:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
  - then count the number of operations executed as function of size of input

Counting Operations(2)

```
def mysum(x):
        total = 0
        for i in range(x+1):
                total += i
        return total
```



1+3x operation

Counting Operations(3)

- Running time varies implementation (Programming Languages)
- Running time does not varies between computers.
- Time varies for different inputs and can come up with a relationship between inputs and the count.
- no real definition of which operations to count.
- BETTER, BUT STILL…

Counting Operations(4)

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

What is the number of steps it will take to run linearSearch in the best case?
n: the number of elements in the list L.

Solution:
In the best case scenario, L is an empty list. Thus one step is taken: return False.

Counting Operations(5)

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

What is the number of steps it will take to run linearSearch in the worst case?
n: the number of elements in the list L.

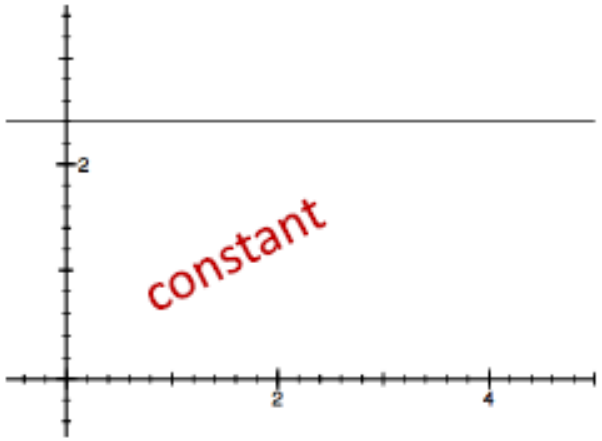
Solution: 2*n + 1

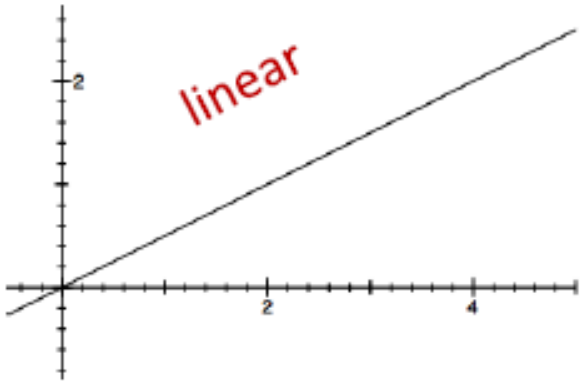
- In the worst case scenario, element is not present in L. Thus we go through the for loop n times. This means we execute assignment of e to each element of L (this takes place in the line for e in L) to enter the for loop, and also execute the check
  if e == x:
- once each for every element. So this is 2*n steps. Finally at the end of the for loop, we execute the return statement one time.

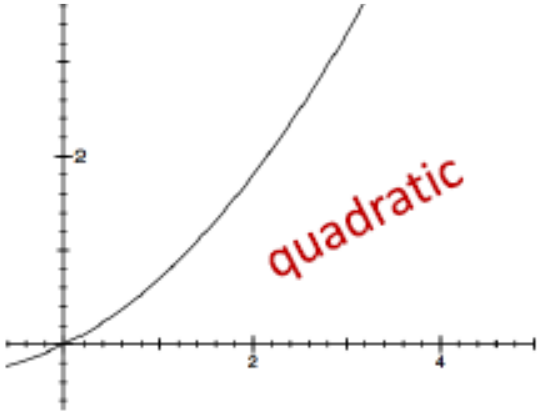
Orders of Growth

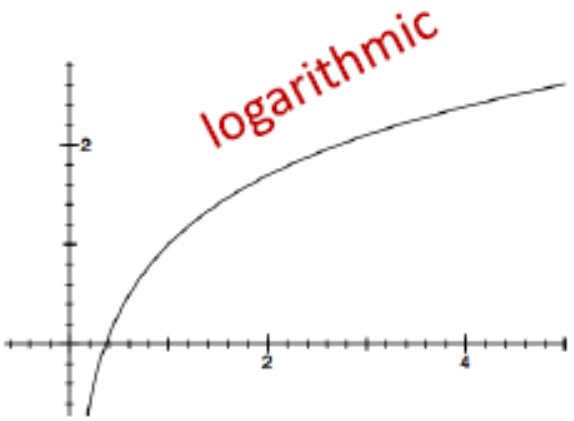
- Goals:
  - want to evaluate program's efficiency when input is very big
  - want to express the growth of program's run time as input size grows
  - want to put an upper bound on growth
  - do not need to be precise: "order of" not "exact" growth

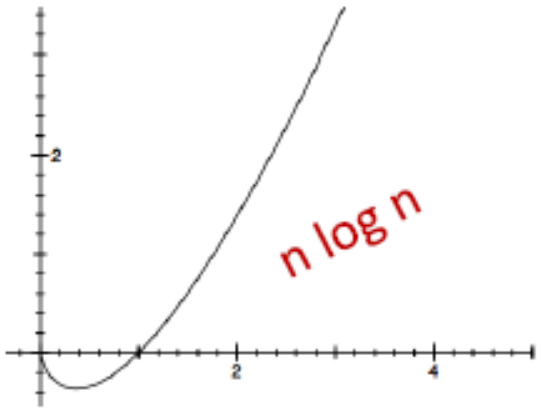
Types of Orders of Growth

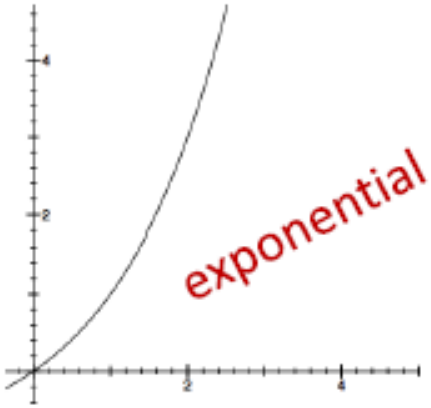










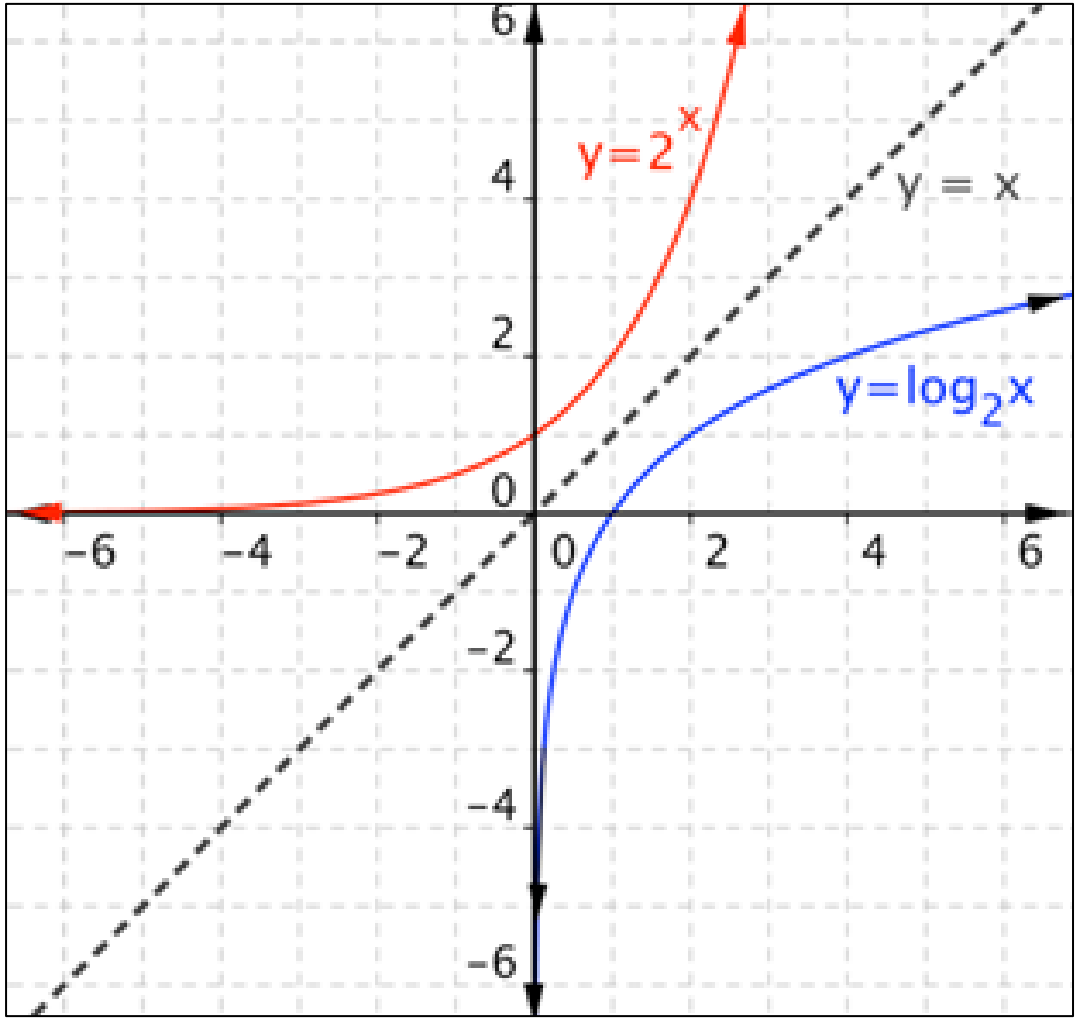


Edx-MIT: Introduction_to_Computer_Science_and_Programming_Using_Python

log2

- Recall is the # of times you divide by 2 until you get down to 1

Logarithmic form		Exponential form
$\log_2(8) = 3$	$\iff$	$2^3 = 8$
$\log_3(81) = 4$	$\iff$	$3^4 = 81$
$\log_5(25) = 2$	$\iff$	$5^2 = 25$



Order of Growth: Big O-Notation

- Evaluate program's efficiency when input is very big
- Express the growth of program's run time as input size grows
- Put an upper bound on growth

Note: not precise: "order of" not "exact" growth

What is the number of steps in the best case?

```
def program1(n):
    total = 0                          → 1
    for          i          in         → 1000
range(1000):                           → 2000
        total += i
                                       → 1

    while n > 0:
        n -= 1                         → 1
        total += n
                                       ---------
                                       3003

    return total
```

In the best case scenario, n is less than or equal to 0.

1. **We first execute the assignment total = 0**
2. **for i in range(1000) loop. This loop is executed 1000 times and has three steps**
   a) **one for the assignment of i each time through the loop**
   b) **two for the += operation on each iteration.**
3. **We next check if n > 0 - it is not so we do not enter the loop.**
4. **Adding one more step for the return statement.**

in the best case we execute 1 + 3*1000 + 1 + 1 = 3003 steps.

What is the number of steps in the worst case?

```
def program1(n):
    total = 0                          # 1
    for i in range(1000):              # 1000
        total += i                     # 2000
                                       # 1n+1
    while n > 0:                       # 2n
        n -= 1                         # 2n
        total += n                     # 1
                                       # 5n+3003
    return total
```

In the worst case scenario, n is a large positive number.

We first execute the assignment total = 0

1. **for i in range(1000) loop. This loop is executed 1000 times and has three steps**
   a) **one for the assignment of i each time through the loop**
   b) **two for the += operation on each iteration.**
2. **the second loop (while n > 0) n times. This loop has five step**
   a) **conditional check, n > 0**
   b) **and two each for the -= and += When we finally get to the point where n = 0.**
3. **We next check if n > 0 - it is not so we do not enter the loop.**
4. **Adding one more step for the return statement.**

in the worst case = 5*n + 3003 steps.

Complexity=O(n)

Big O Example(1)

```
def fact_iter(n):
        """assumes n an int>= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

- computes factorial ➔5n+2
- worst case asymptotic complexity *O(n)*:
  - ignore additive constants
  - ignore multiplicative constants
  - focus on dominant terms

Big O Example(2)

1) 600
2) $\mathbf{log(n) + 4}$
3) $\mathbf{log(n) + n + 4}$
4) $\mathbf{0.0001 * n * log(n) + 300n}$
5) $\mathbf{n^2 + 2n + 2}$
6) $\mathbf{n^2 + 100000n + 31000}$
7) $\mathbf{2n^{30} + 3^n}$

1) O(1) constant
2) O $(\log n)$ logarithmic
3) O(n) linear
4) O(nlogn) log-linear
5) O(n^2) polynomial
6) O(n^2) polynomial
7) O(3^n) exponential

Big O Example(3)

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

Complexity: O(n^2)

```
for i inrange(n):
    for j inrange(n):
        print('a')
```

Complexity: O(n^2)

```
for i inrange(100):
    print('a')
```

Complexity: O(1)

Growth of Functions

- Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it.
- For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.
- Insertion sort's worst-case running time as $an^2 + bn + c$, for some constants a, b, and c. By writing that insertion sort's running time is $\Theta(n^2)$.

Where Does Notation Come From?

Big-Oh is a member of a family of notations invented by Paul Bachmann, Edmund Landau(German mathematician), and others, collectively called Bachmann–Landau notation or asymptotic notation

"On the basis of the issues discussed here, I propose that members of SIGACT, and editors of compter science and mathematics journals, adopt the O, Ω, and Θ notations as defined above, unless a better alternative can be found reasonably soon".

-D. E. Knuth, "Big Omicron and Big Omega and Big Theta", SIGACT News, 1976. Reprinted in "Selected Papers on Analysis of Algorithms."

Asymptotic notation

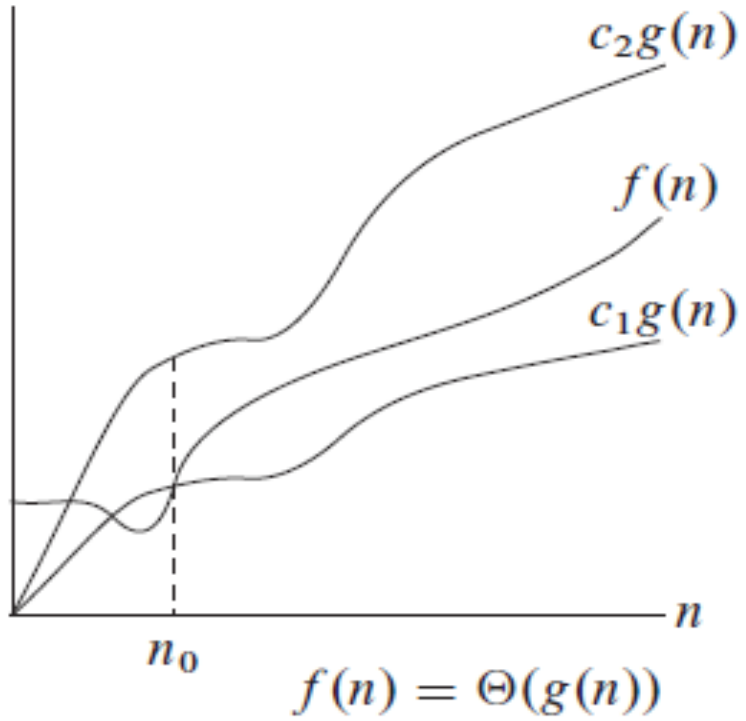
- Asymptotic notation concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions.
  - Domains are the set of natural numbers $\mathcal{N} = \{0,1,2, \ldots\ldots\}$.
- Asymptotic notations are convenient for describing the worst-case running-time function T(n), which usually is defined only on integer input sizes.
- we might extend the notation to the domain of real numbers or, alternatively, restrict it to a subset of the natural numbers.

Θ-notation

- $\Theta(g(n))$ the set of functions
- $\Theta(g(n)) = \{there\ exist\ positive\ constants\ c_1, c_2, and n_0 such\ that\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) for\ all\ n \geq n_0\}$

Θ -notation bounds a function to within constant factors. We write f(n)= $\Theta(g(n))$if there exist positive constants $c_1, c_2, and n_0$ such that at and to the right of n_0, the value of $f(n)$ always lies between (sandwiched) $c_1 g(n)$ and $c_2 g(n)$ inclusive.

For all values of n ≥ n_0, the value of f(n) lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$

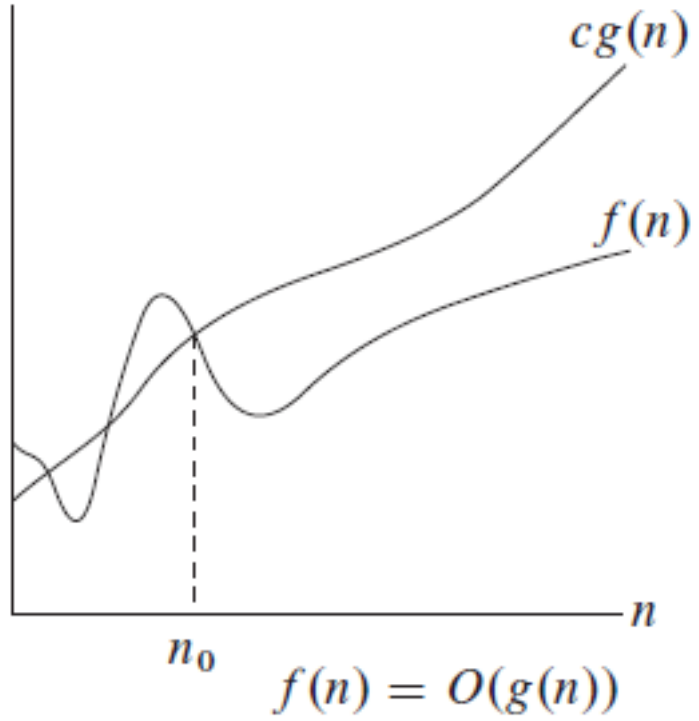


$f(n) = \Theta(g(n))$ (asymptotically equal)
$f(n) \in \Theta(g(n))$
$g(n)$ is an ***asymptotically tight bound for $f(n)$***.

O-notation

- $O(g(n)) = \{f(n)|$there exist positive constants c and$n_0$such that $0 \leq f(n) \leq cg(n) \forall\, n \geq n_0\}$

O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

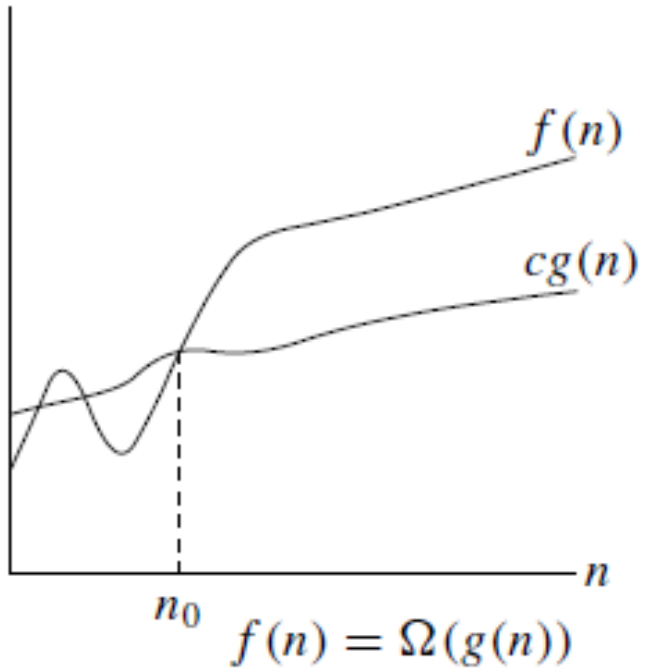


Ω-notation

- $\Omega(g(n)) = \{there\ exist\ positive\ constants\ c\ and n_0 such\ that$

$$0 \leq cg(n) \leq f(n) \forall\ n \geq n_0\}$$

Ω -notation gives a lower bound for a function to within a constant factor. We write *f (n)* = Ω *(g(n))* if there are positive constants *n*0 and *c* such that to the right of *n*0, the value of *f (n)* always lies on or above *cg(n)*.



Pseudo Code

- Pseudocode is a plain language description of the steps in an algorithm or another system. Pseudocode often uses structural conventions of a normal programming language, but is intended for human reading rather than machine reading. It typically omits details that are essential for machine understanding of the algorithm, such as variable declarations and language-specific code.

https://en.wikipedia.org/wiki/Pseudocode

Exact versus Approximate algorithms

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- http://www.math.uwaterloo.ca/tsp/index.html

- Can we use Exact Solution?

Sorting Algorithms

The problem of sorting

- *Input:* array A[1…n] of numbers.
- *Output:* permutation B[1…n] of A such that $B[1] \leq B[2] \leq \ldots \leq B[n]$.
- e.g. A = [7, 2, 5, 5, 9.6] → B = [2, 5, 5, 7, 9.6]
- How can we do it efficiently ?

Bubble Sort

Bubble Sort- Gif

https://commons.wikimedia.org/wiki/File:Bubble-sort.gif

Bubble Sort- Dance

https://www.youtube.com/watch?v=lyZQPjUT5B4

Bubble Sort-Algorithm

```
Repeat until no more swaps
    for all elements of list
        if list[i] > list[i+1]
            swap(list[i], list[i+1])
        end if
    end for
```

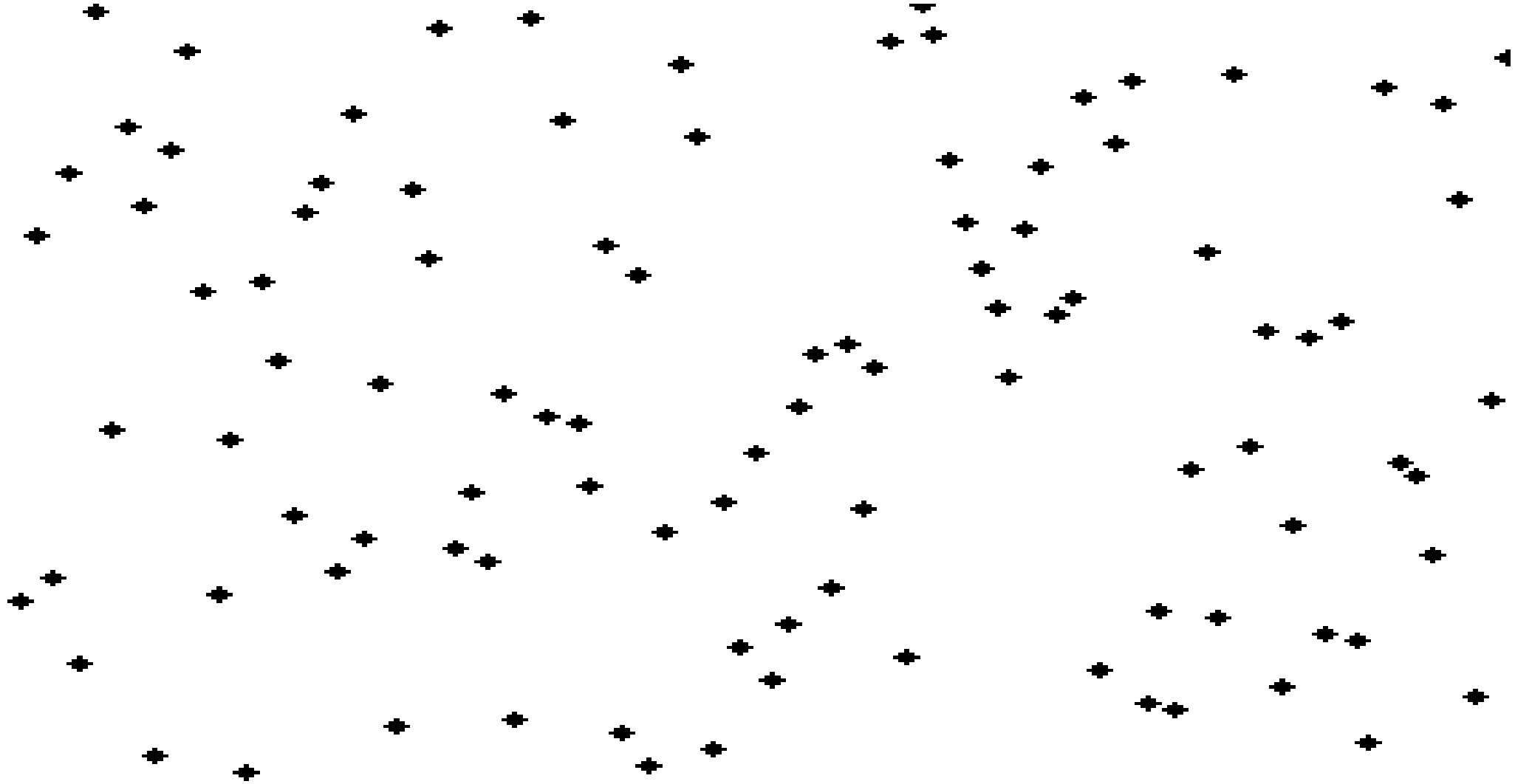
Bubble Sort-Code

```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        print(L)
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

Bubble Sort-Complexity Analysis

- In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,
- (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
- Sum = n(n-1)/2
- The time complexity of Bubble Sort is O(n^2).
- The space complexity for Bubble Sort is O(1), because only a single additional memory space is required i.e. for temp variable.
- Also, the best case time complexity will be O(n), it is when the list is already sorted.

Bubble Sort-Animation



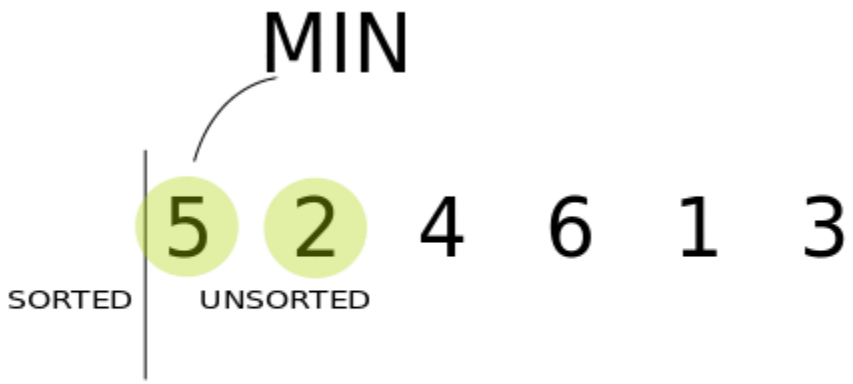
Obama and Bubble Sort

- Obama in 2007 at Google - Eric Schmidt asked Obama how to sort 1 million integers as a laugh line, but Obama shocked him by answering "I think the bubble sort would be the wrong way to go."
  - What is the most efficient way to sort a million 32-bit integers? OBAMA: Well...
  - SCHMIDT: Maybe--I'm sorry...
  - OBAMA: No, no, no, no. I think--I think the bubble sort would be the wrong way to go.

Obama and Bubble Sort

Selection Sort

Selection Sort-gif



https://codepumpkin.com/selection-sort-algorithms/

Selection Sort-dance

https://www.youtube.com/watch?v=Ns4TPTC8whw

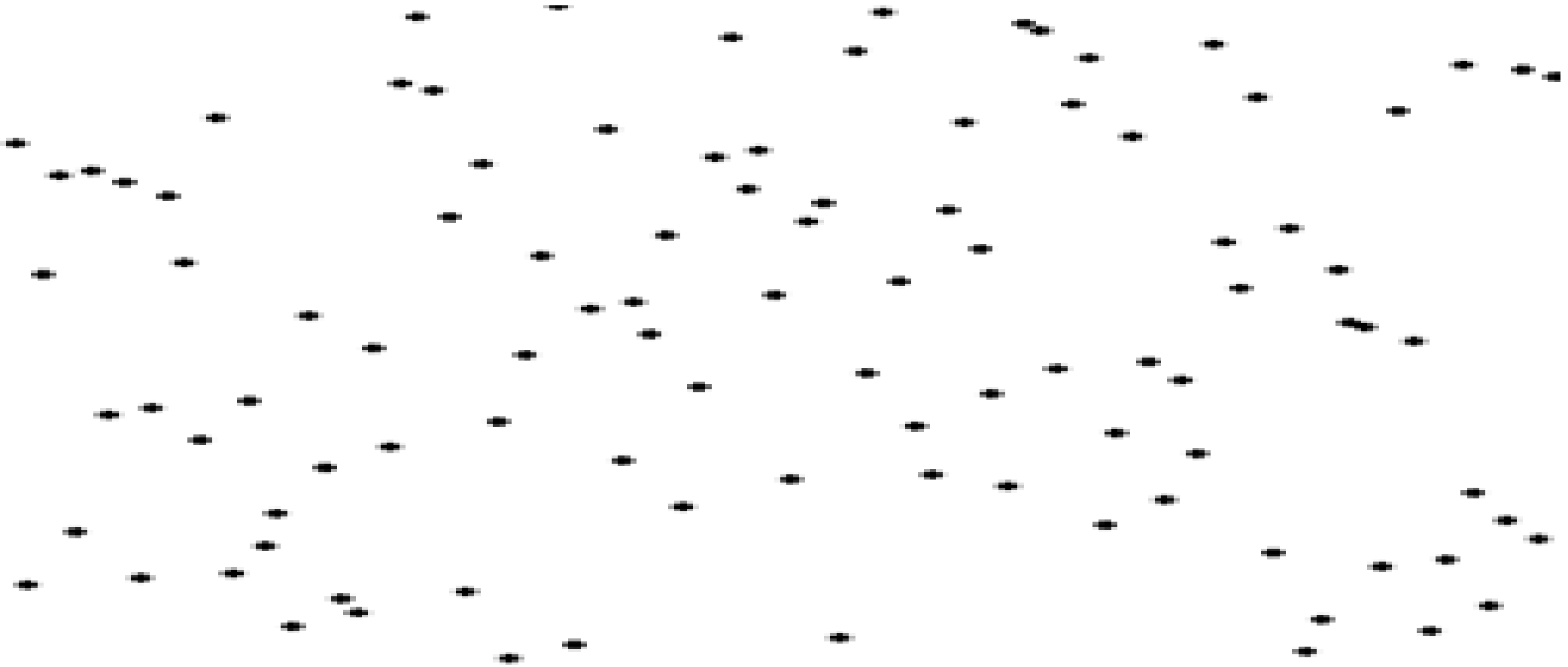
Selection Sort-Algorithm

1. Choose minimum element, swap it with element at index 0
2. in remaining sublist, extract minimum element swap it with the element at index 1
3. and so on

Selection Sort-Complexity Analysis

- Worst Case Time Complexity [Big-O]: O(n^2)
- Best Case Time Complexity [Big-omega]: $\Omega(n^2)$
- Average Time Complexity [Big-theta]: $\Theta(n^2)$
- Space Complexity: O(1)

Selection Sort-Animation



Insertion Sort

Insertion Sort-gif

Programming paradigms

- The imperative paradigm is the oldest and most popular paradigm.
- Program consists of sequence of statements that change the program state
- Based on the von Neumann architecture of computers
  - Machine languages
  - FORTRAN
  - COBOL
  - C
- Procedural programming is a refinement that makes it easier to write complex programs
- Abstract Data Types is a further extension of imperative programming
  - Data and operations are encapsulated into a bundle (information hiding)
  - This hides the underlying representation and implementation
- Object-oriented paradigm extends ADTs Classes are blueprints for objects that encapsulae both data and operations Objects exchange messages Provides encapsulation, information hiding, inheritance, and dynamic dispatchin

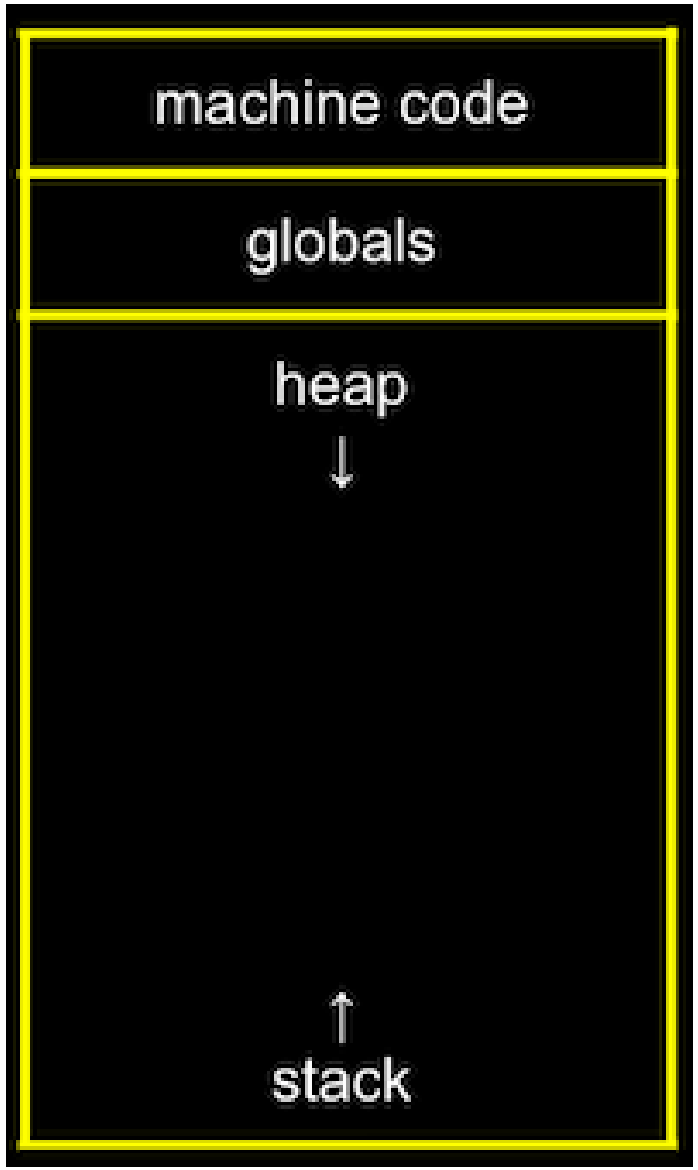
Recursion

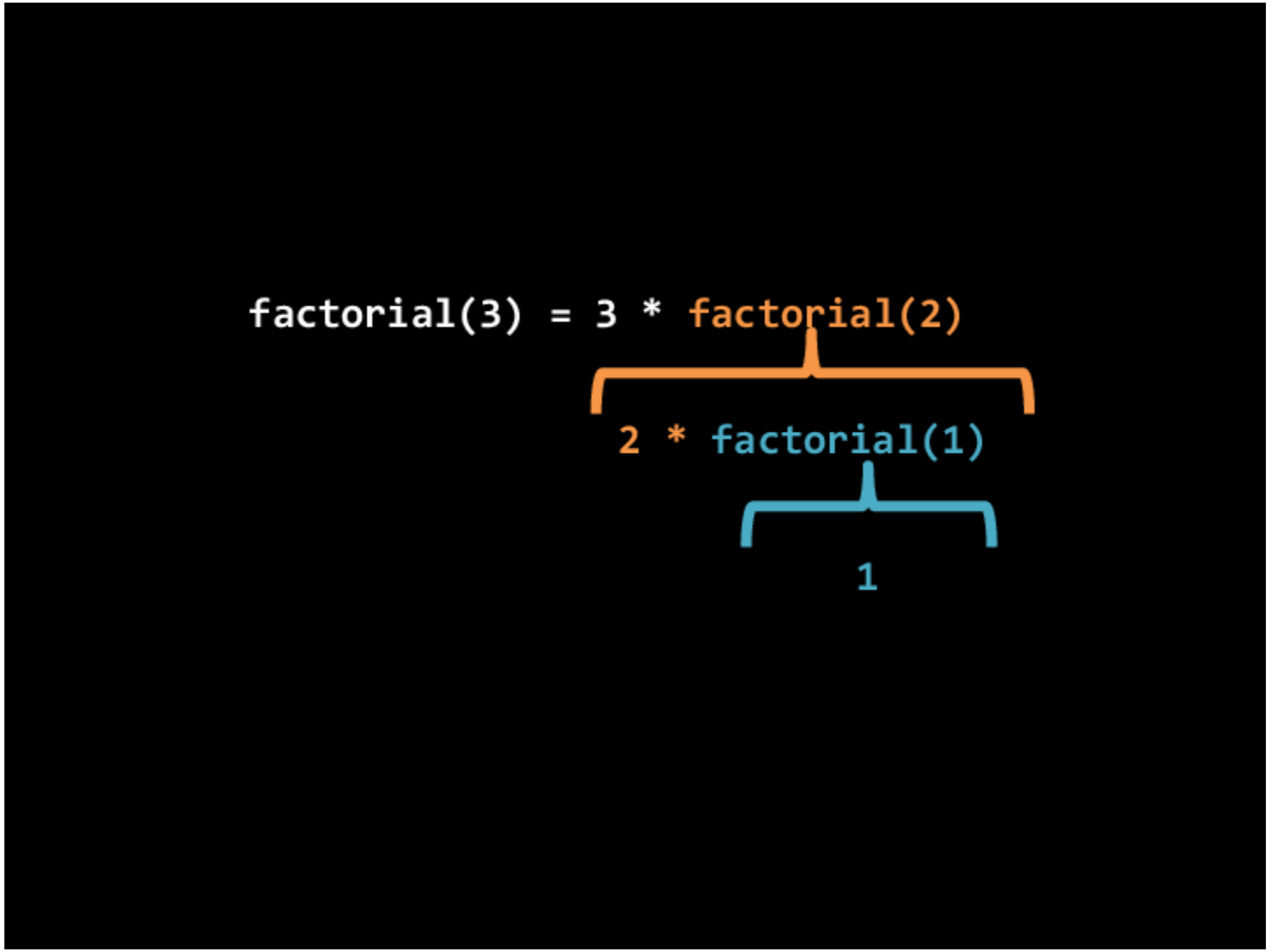
- Recursion occurs when a function calls itself repeatedly until it reaches the termination Condition.
- Why?
  - Recursion provides a methodology to get compact, simple and yet elegant Algorithms
  - More natural.
- Cons.
  - Recursive code is slower than iterative code, since it not only runs the program but must also invoke stack memory(use tail recursion).
  - Uses more memory: Each call saves the previous step in the call stack until the recursive process is complete. This means recursive solutions use more memory than iterative ones.
  - Maximum recursion depth: Python has a limited call stack that supports only 1000 nested steps. While this may seem like a lot, it becomes an issue when dealing with large structures like lists and arrays. This can be overridden at your own rise withsys.setrecursionlimit(1500).
- Components of recursion
  - Base case
  - Recursive calls

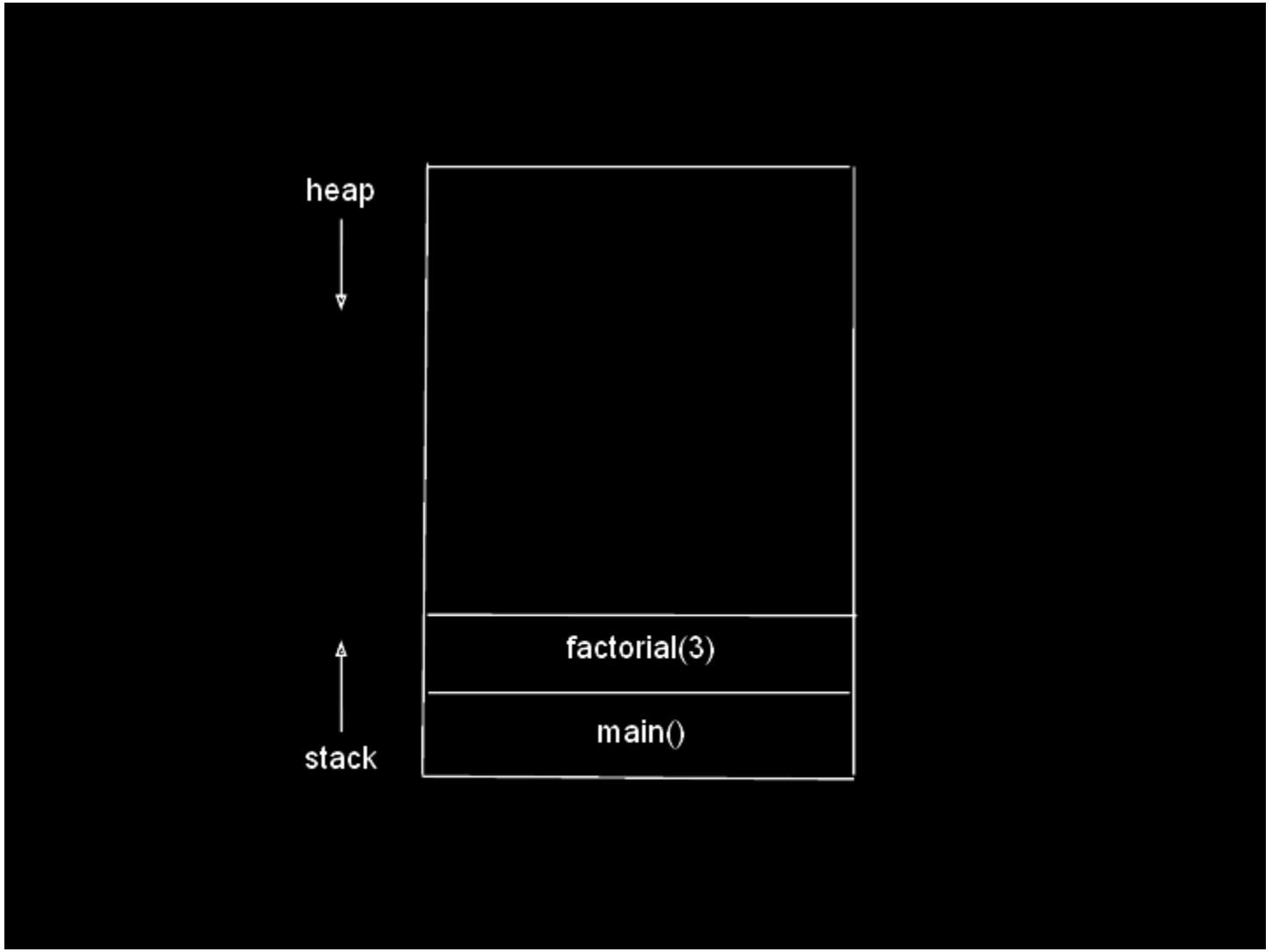
Recursion Template

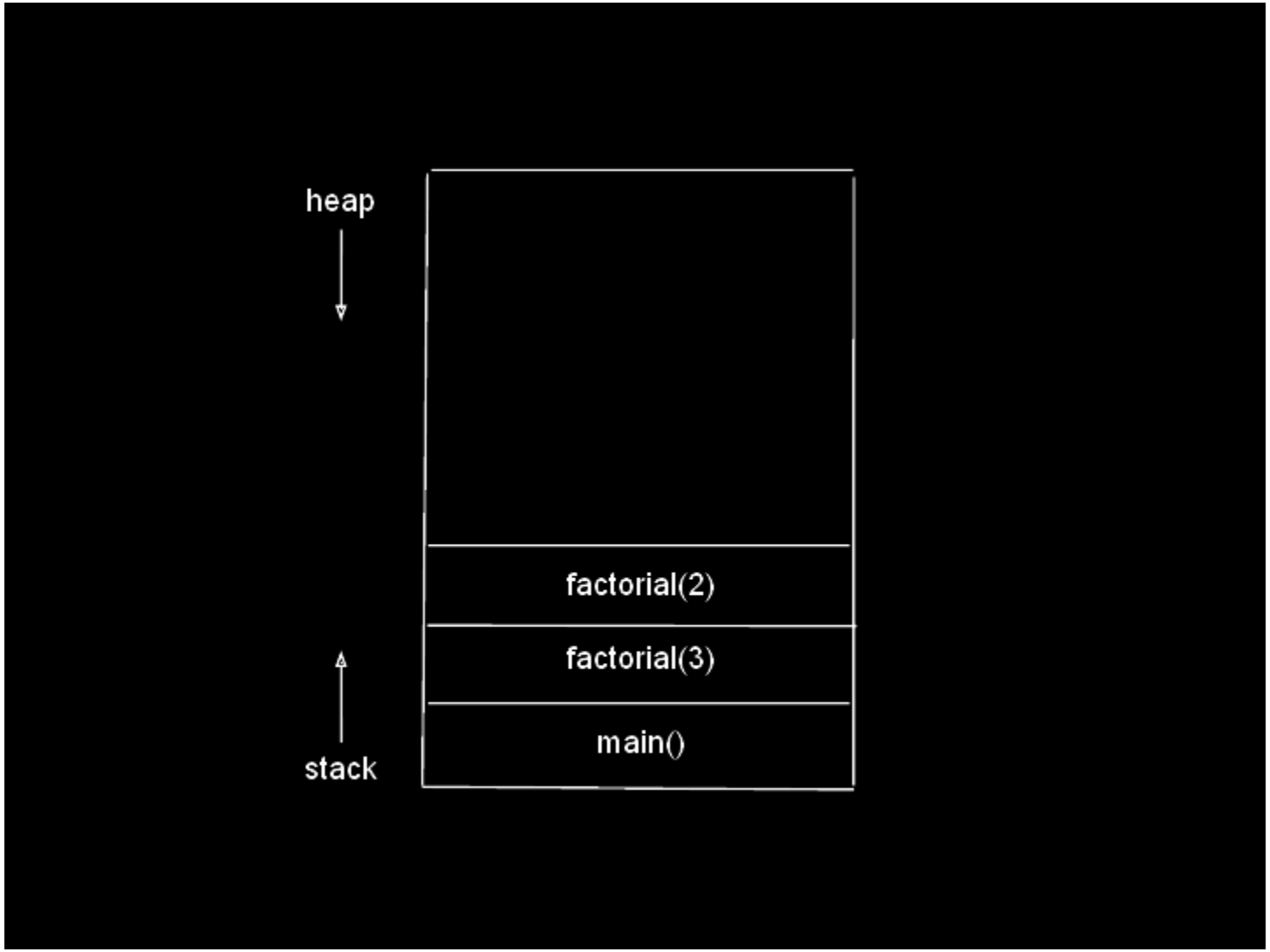
```
def RecursiveFunction() :
  # Base Case
  if <baseCaseCondition> :
    <return some base case value>


  # Recursive Case
  else :
    <return(recursive call and any other task)>
```

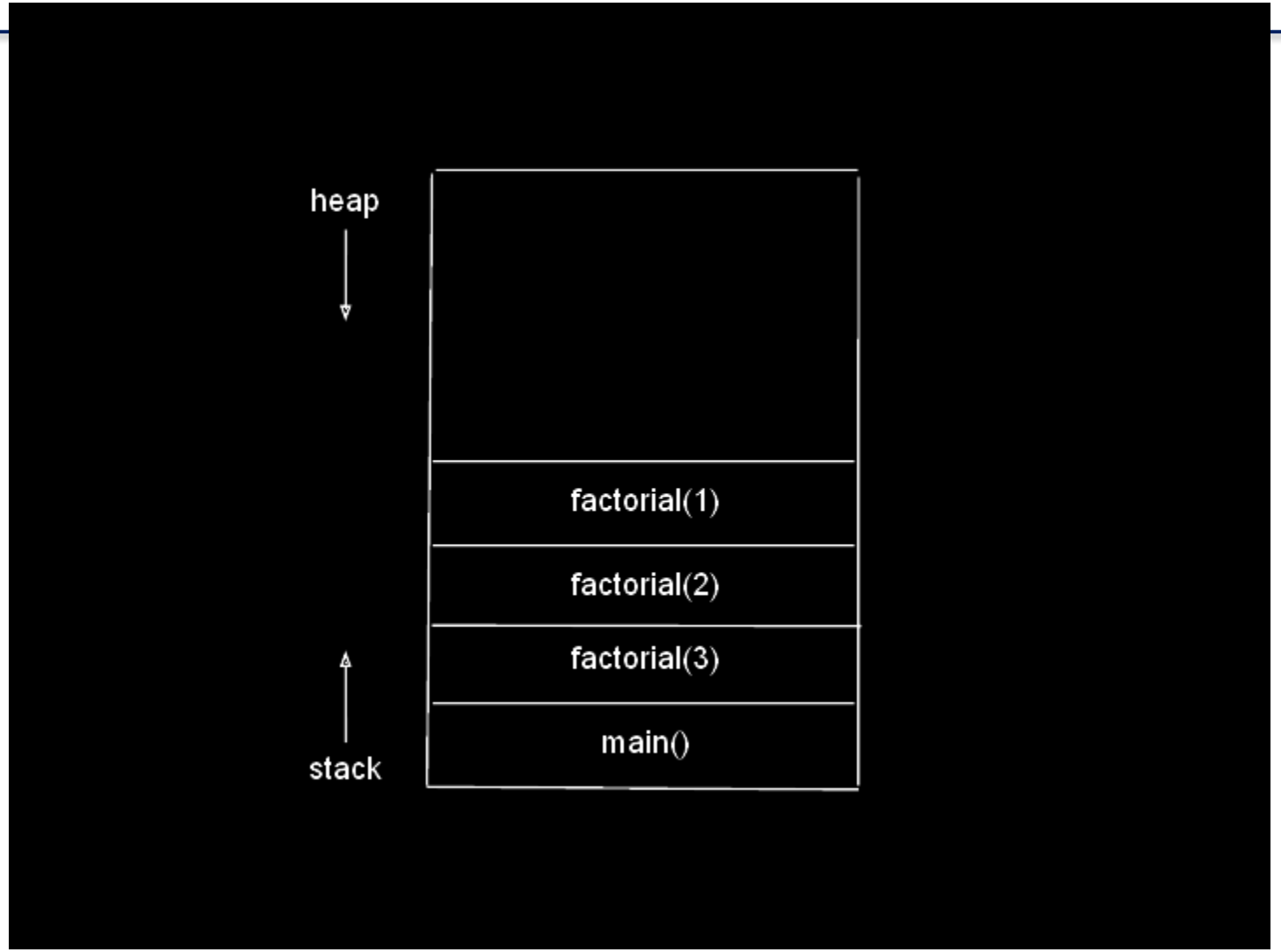
Memory layout

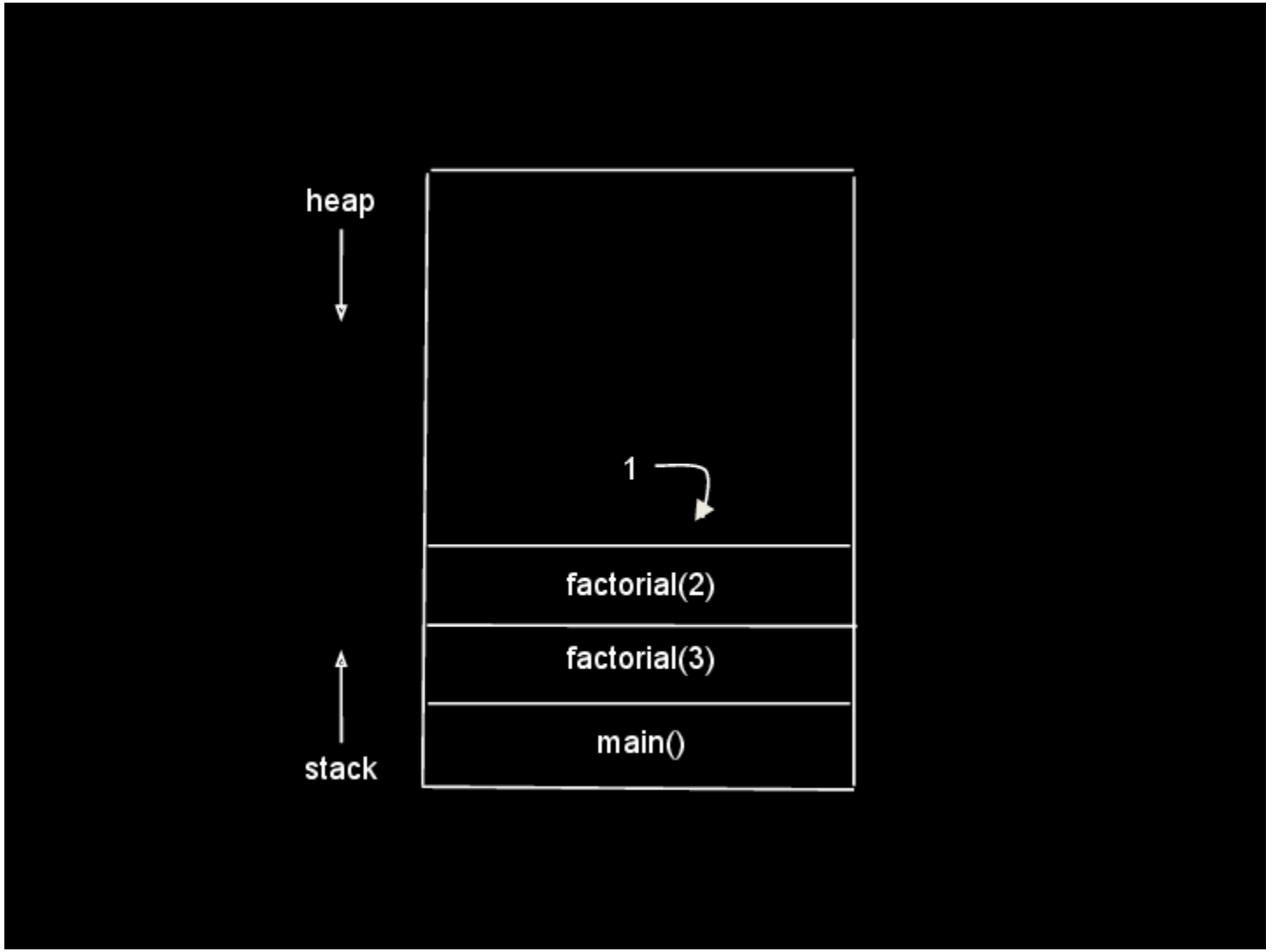


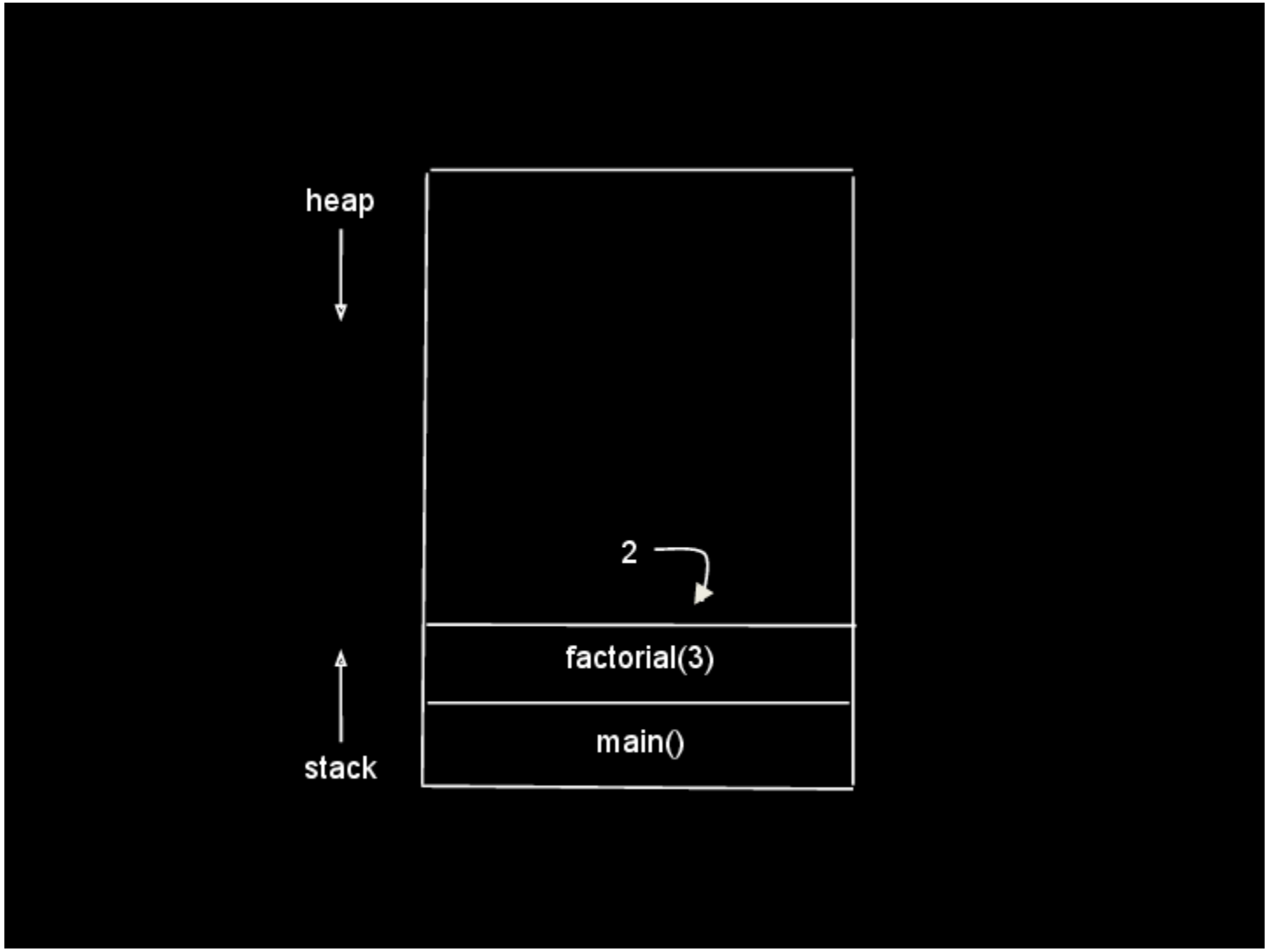

factorial(3) = 3 * factorial(2)
2 * factorial(1)
1

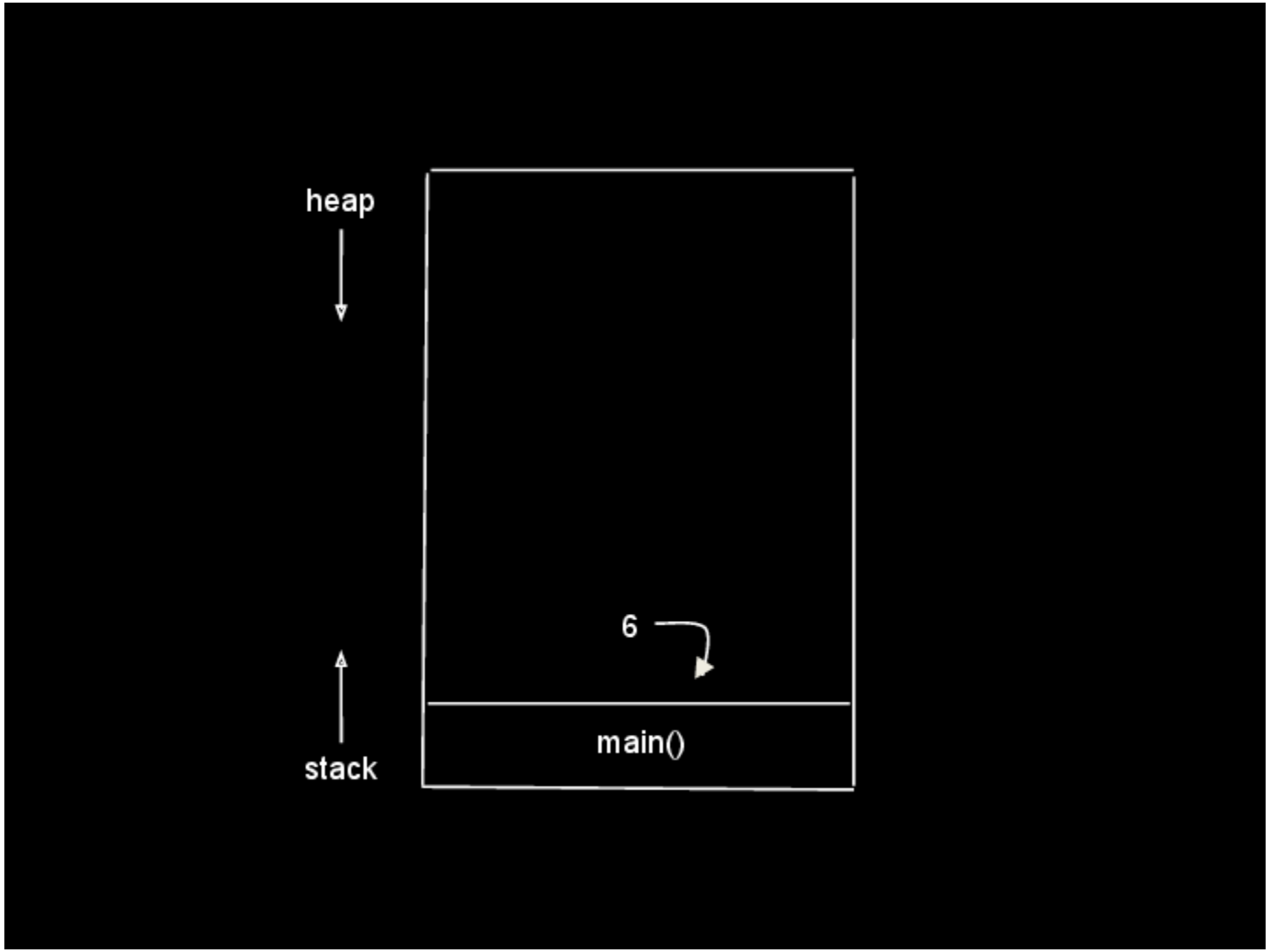

heap
factorial(3)
main()
stack


heap
factorial(2)
factorial(3)
main()
stack


heap
factorial(1)
factorial(2)
factorial(3)
main()
stack


heap
1
factorial(2)
factorial(3)
main()
stack


heap
2
factorial(3)
main()
stack


heap
6
main()
stack

Recursion

- Some languages lack recursion: Fortran77, Assembler
- Some languages allow recursion, but aren’t very efficient with it: C++, Java
- Languages optimized for recursion

Terminal

- Terminal recursion is a principle that solves these time and memory problems
- a function is terminal recursive if all its return paths are of the form return f(...);.
- Terminal recursion is effective because it simply allows one to branch former arguments into new arguments by
- performing various expression evaluations on parameters only. Therefore there
- is no need to use the stack function calls, and no stack overflow problems are
- occurring when using terminal recursion
- https://recursion.vercel.app/

Tail Recursion

- Tail call The biggest advantage of using tail calls is that they allow you to do extensive operations without exceeding the call stack. This makes it possible to do a lot of work in constant space without running into out of memory exceptions; this happens because the frame for the currently executing function is re-used by the newly-executed function call.
- It is possible to write the same function using tail call recursion and this will allow you to do it for arbitrarily large values. The space optimization is from O(n)to 0(1) since every new function call reuses existing frames rather than pushing stuff on the stack.

Tail Recursion

- The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use

Greatest Common Divisors

Definition

- For integers, a and b, their greatest common divisor or gcd(a, b) is the largest integer d so that d divides both a and b.
- Input: Integers $a, b \geq 0$.
- Output: gcd(a, b).

Greatest Common Divisors (GCD)

- Lemma
- Let a’ be the remainder when a is divided by b, then
- gcd(a, b) = gcd(a’, b) = gcd(b, a’).

Euclidean Algorithm

- gcd(3918848, 1653264)
- =gcd(1653264, 612320)
- =gcd(612320, 428624)
- =gcd(428624, 183696)
- =gcd(183696, 61232)
- =gcd(61232, 0)
- =61232.
- Each step reduces the size of numbers by about a factor of 2.
- Takes about log(ab) steps.
- GCDs of 100 digit numbers takes about
- 600 steps.
- Each step a single division.