

Lesson 3

Data Structures and Algorithms DSA

In this lesson we will talk about:

- ▶ data structures
- ▶ algorithm design
- ▶ algorithm efficiency
- ▶ searching and sorting algorithms

Data Structures

What is a data structure?

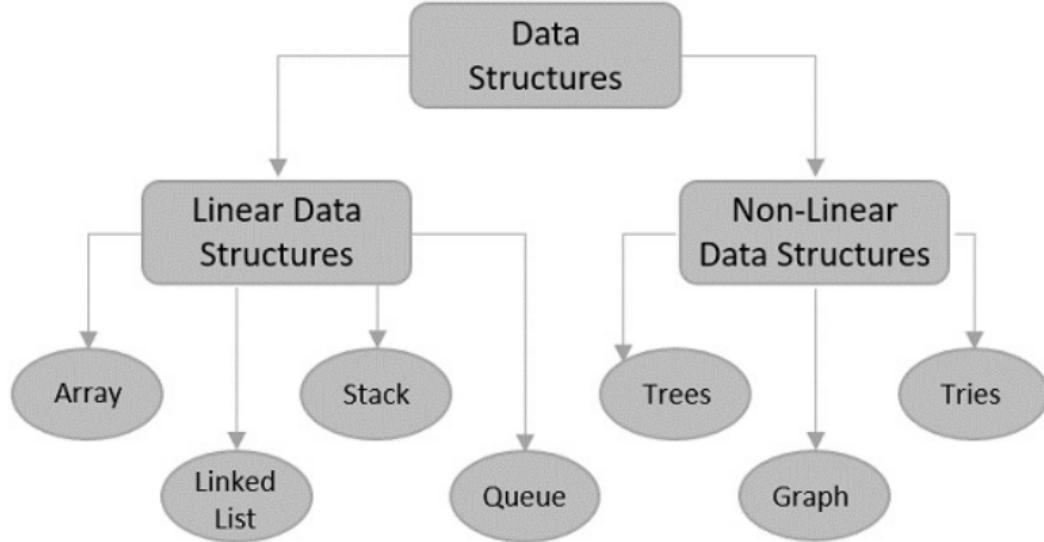
In computer science, a **data structure** is a data organization and storage format that is usually chosen for efficient access to data.^{[1][2][3]} More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data,^[4] i.e., it is an algebraic structure about data.

Data Structures

- ▶ How do we **organize** data
- ▶ For a very **efficient** access

It's a collection of data values, and the relationships among these values

Data Structures



Linear Data Structures: Arrays, Queues, Stacks

- ▶ Elements are arranged sequentially, one after the other
- ▶ The first element added will be the first one to be accessed or removed, and the last element added will be the last one to be accessed or removed
- ▶ Can have either fixed or dynamic sizes
- ▶ Offer very efficient data access

Non-linear Data Structures: Trees, Graphs

- ▶ Elements are arranged hierarchical
- ▶ We can't traverse all the elements in a single run only
- ▶ There are multiple levels which we must traverse
- ▶ It is more difficult to implement

Data structures

- ▶ Sets
- ▶ Arrays and Matrices
- ▶ Stacks
- ▶ Queues
- ▶ Linked lists
- ▶ Trees
- ▶ Graphs

Sets

Sets

A set is usually a **collection** of different things, fixed in size. Sets can also change size, usually when an algorithm will perform modifications against the set. We call these dynamic sets. These sets can change in size, grow or shrink, basically change over the time.

Lesson 3

Sets

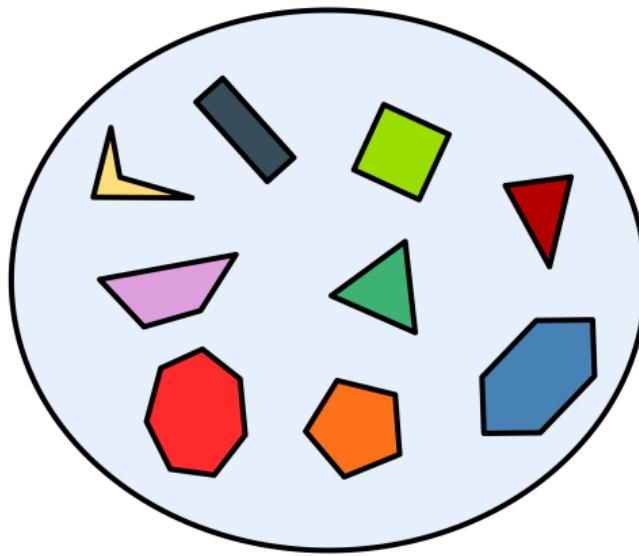
But what is a set?

Sets

The basic, fundamental data structure: {1,2,4,51,9}

- ▶ mathematical set
- ▶ unchanging, unique elements, no duplicates
- ▶ contains a fixed number of elements: finite set
- ▶ or it can contain an infinite number of elements

For example a set of polygons



Sets

A set is a **mathematical model** of a collection of different things. A set contains elements or members, which can be mathematical objects of any kind numbers, symbols, points in space, lines, other geometrical shapes, variables, or even other sets.

Sets, examples

- ▶ $\{\text{white, blue, red, yellow}\}$
- ▶ The empty set $\{\}$
- ▶ Natural numbers: $\mathbb{N} = \{0, 1, 2, 3, \dots\}$
- ▶ Natural numbers except 0: $\mathbb{N}^* = \{1, 2, 3, \dots\}$
- ▶ Integers: $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
- ▶ Positive integers: $\mathbb{Z}_+ = \{0, 1, 2, 3, \dots\}$

Lesson 3

Sets

{1,2,3,4}

Defines a list of elements, using a simple enumeration notation (Roster notation) between curly brackets, separated by commas.

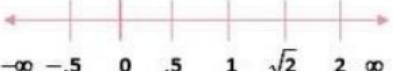
Basic Operations on Sets

- ▶ Insert - add a new element to a set
- ▶ Delete - remove an element from a set
- ▶ Test - if element X belongs to a set or not

A dynamic set which supports all these basic operations: **a dictionary**

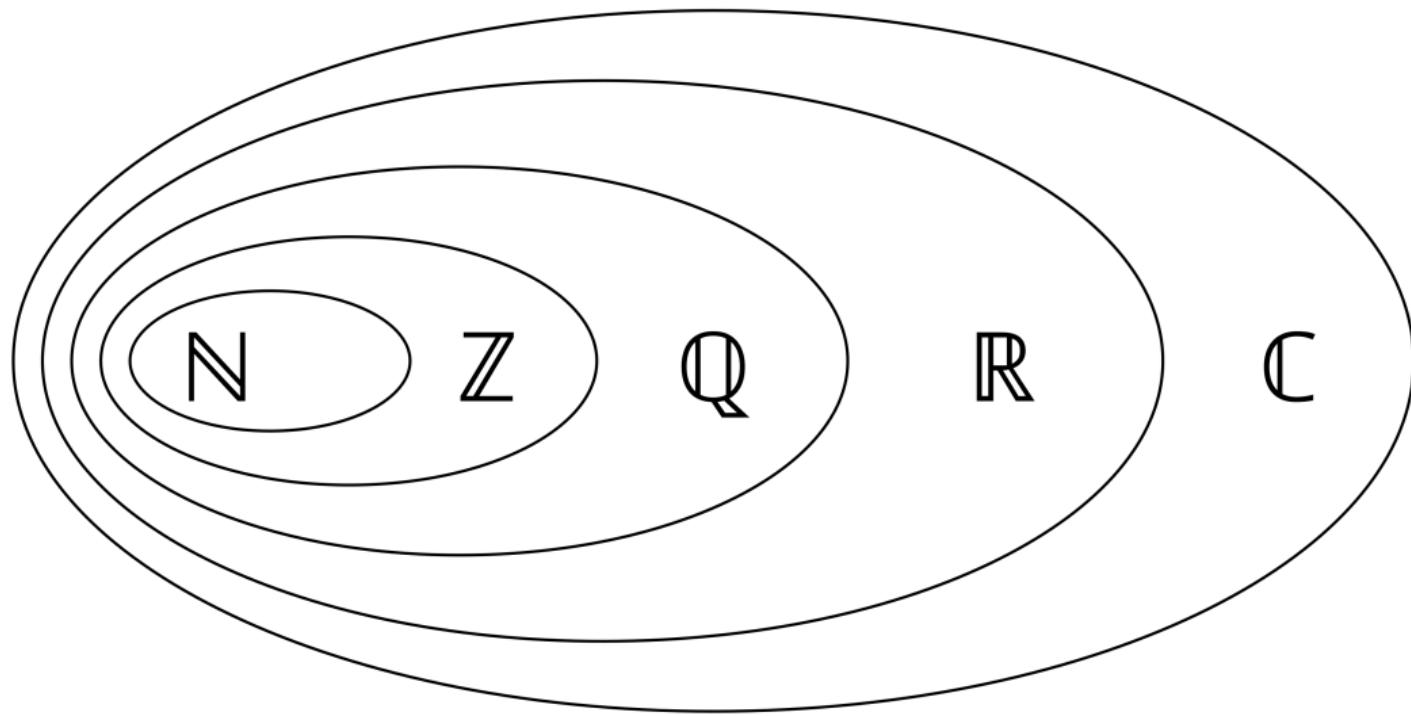
Lesson 3

Sets

Natural Numbers (Counting Numbers) (N)	Numbers you use for counting: 1, 2, 3 ...	It's "natural" to count on your fingers: 1, 2, 3,
Whole Numbers	The natural numbers, plus 0: 0, 1, 2, 3 ...	The word "whole" has an "o" in it, so include 0.
Integers (Z)	Whole numbers, their opposites (negatives), plus 0: ... -2, -1, 0, 1, 2 ...	Integers can be separated into negative, 0, and positive numbers.
Rationals (Q)	Integers and all fractions, positive and negative, formed from integers. These include repeating fractions, such as $\frac{1}{3}$, or .33333.. or $\bar{3}$.	The word "rational" is a derivation of "ratio", and rational numbers are numbers that can be written as a ratio of two integers. "Q" stands for quotient.
Irrationals	Numbers that cannot be expressed as a fraction, such as π , $\sqrt{2}$, e. (We'll learn about these later).	If something is "irrational", it's not easy to explain or understand.
Real Numbers (R)	Rational numbers and Irrational Numbers. The real number system can be represented on a number line: 	If a number exists on a number line that you can see, it must be "real". Note that the "smallest" real number is negative (-) infinity ($-\infty$), and the largest real number is infinity (∞). We can never really get to these "numbers" ($-\infty$ and ∞), but we can indicate them as the "end" of the real numbers.
Complex Numbers (C)	Real numbers, plus imaginary numbers (concept only, such as $\sqrt{-2}$).	"Imaginary" numbers are difficult to imagine, since they are so "complex".

Lesson 3

Sets



Advantages

Perform operations on a collection of elements in a very
efficient and **organized** manner

Conclusions

Sets are basic, fundamental data structures, with:

- ▶ unique elements
- ▶ no duplicates
- ▶ unchanging
- ▶ fixed or infinite number of elements

I'm confused. Does it mean a set is similar to a Python set? Or what is the difference?

Sets vs. Python Set

In computer science (CS), a set is an abstract data type that can store unique values, without any particular order. It is a computer implementation of the mathematical concept of a finite set.

Mathematical vs. Python Sets

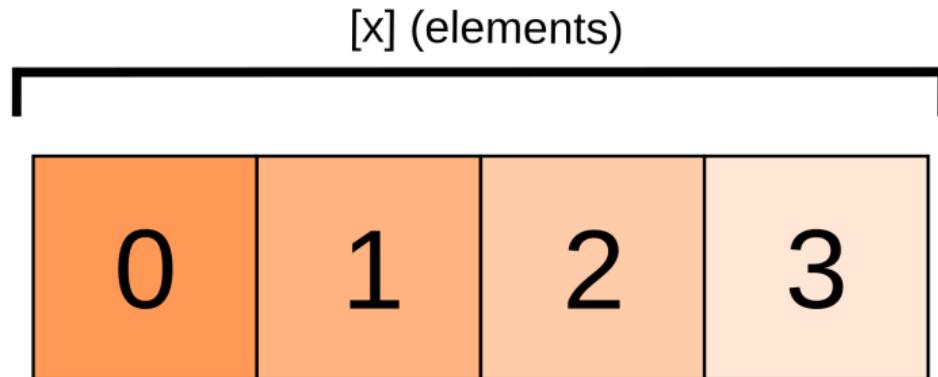
- ▶ Mathematical finite set: $\{1,2,3,4\}$
- ▶ Python set: $S = \{1,2,3,4\}$

Arrays

Arrays

In computer science, an **array** is a data structure consisting of a collection of elements, each identified by an **index** or a **key**. The simplest type of such data structure is a linear array, the one-dimensional array.

Typical "1 Dimensional" array



Element indexes are typically defined in the format [x]
[x] being the number of elements
For example: this array could be defined as `array[4]`

Arrays

Arrays are among the oldest and most important data structures, and are used by almost every program and programming language. They are also used to implement many other data structures, such as lists.

Arrays

Arrays are useful because the element indices can be computed at **run time**. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation.

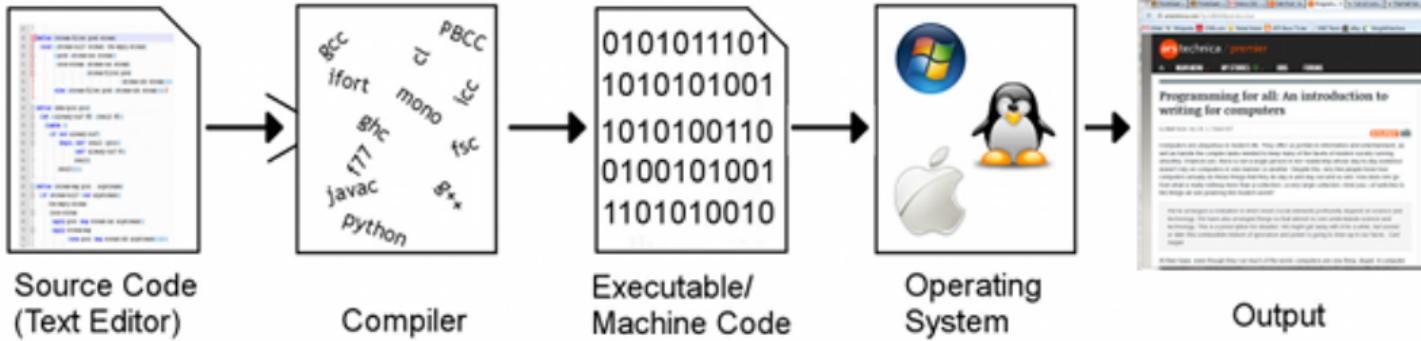
Run-time?

Runtime, run time, or execution time is the final phase of a computer program's life cycle, in which the code is being executed on the computer's central processing unit (CPU) as machine code. In other words, "runtime" is the running phase of a program.

Lesson 3

Arrays

Remember this? From source code to executable



Basic Array Operations

- ▶ traversal of an array
- ▶ access element X in an array
- ▶ searching element X in an array
- ▶ sorting an array

Example 1: Traversing the array A

```
1: procedure GETARRAY( $A$ )      ▷ Returns the max value in  $A$ 
2:      $L \leftarrow \text{length}(A)$ 
3:     for  $i=0$  to  $L-1$  do
4:         print  $A[i]$ 
5:     end for
6: end procedure
```

Traversing the array

A = [1, 2, 3, 4, 5, 6, 7,8,9]

```
# Traversing the array
for element in A:
    print(element, end="-")}
```

Traversing the array, version 2

```
A = [1, 2, 3, 4, 5, 6, 7,8,9]
```

```
# Traversing the array
for i in range(len(A)):
    print(A[i], end=',')
```

Example 2: Find the max value in the array A

```
1: procedure MAXARRAY(A)      ▷ Returns the max value in A
2:   N ← length(A)
3:   MAX ← A[0]
4:   for from i=1 to N-1 do
5:     if A[i] > MAX then
6:       MAX = A[i]          ▷ The MAX is A[i]
7:     end if
8:   end for
9:   return MAX
10: end procedure
```

Lesson 3

Arrays

Example 3: Search element X in array A

```
1: procedure SEARCHARRAY(A) ▷ Returns the max value in A
2:   X ← MyElement
3:   N ← length(A)
4:   for from i=0 to N-1 do
5:     if X = A[i] then
6:       return i                                ▷ The index for my match
7:     end if
8:   end for
9:   return -1                                 ▷ otherwise return -1
10: end procedure
```

Search element in array

A = [1, 2, 3, 4, 5, 6, 7,8,9]

```
def find_element(A, n, key):  
    for i in range(n):  
        if A[i] == key:  
            return i  
    return -1
```

There are numerous applications of arrays

- ▶ Storing data in databases. Storing a list of customer names.
- ▶ Traffic Management. Traffic management systems use arrays to track vehicles and their flow. By analyzing data stored in arrays, traffic control centers can implement efficient signal timings and manage congestion effectively.

Lesson 3

Arrays

- ▶ Financial Analysis. It keeps track of various financial instruments, including stocks, bonds, and mutual funds. By organizing data in arrays, companies can perform analyses and make predictions easier
- ▶ Machine Learning. Machine learning algorithms often accept arrays as input, helping to train models and make predictions.

Lesson 3

Matrices

Matrices

What is a matrix?

In mathematics, a matrix (pl.: matrices) is a rectangular array or table of numbers, symbols, or expressions, with elements or entries arranged in rows and columns, which is used to represent a mathematical object or property of such an object.

Lesson 3

Matrices

Matrices

Typical "2 Dimensional" array

[x] (rows)			
00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

[y] (columns)

Element indexes are typically defined in the format [x][y]
[x] being the number of rows
[y] being the number of columns

Matrices

$$\begin{matrix} & 1 & 2 & \dots & n \\ 1 & a_{11} & a_{12} & \dots & a_{1n} \\ 2 & a_{21} & a_{22} & \dots & a_{2n} \\ 3 & a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m & a_{m1} & a_{m2} & \dots & a_{mn} \end{matrix}$$

Matrix multiplication

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Matrix multiplication M x N

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

Basic Matrix Operations

- ▶ access X element in a matrix
- ▶ traversal of a matrix
- ▶ searching a matrix
- ▶ sorting a matrix

Accessing the elements of a matrix

```
A = [[1, 2, 3], [4, 5, 6], [7,8,9]]
```

```
# Accessing certain elements in a matrix
print("1st-element-of-1st-row:", A[0][0])
print("2nd-element-of-the-2nd-row:", A[1][2])
print("2nd-element-of-3rd-row:", A[2][1])
```

Traversing the matrix

```
A = [[1, 2, 3], [4, 5, 6], [7,8,9]]
```

```
# Traversing the matrix
for row in A:
    # Traversing the matrix
    for x in row:
        print(x, end="-")
    print()
```

There are numerous applications of matrices

- ▶ Encryption: Matrices encrypt data into unreadable formats and decode it for secure communication.
- ▶ Computer Graphics: transformations like scaling, rotation, and translation of objects in 2D and 3D graphics
- ▶ Machine Learning: fundamental data structures for neural networks

Lesson 3

Matrices

- ▶ Economics and Business: optimize business operations like supply chains and financial forecasting
- ▶ Navigation Systems: GPS systems use matrices to calculate positions, distances, and directions in 2D and 3D space.
- ▶ Weather Prediction: Matrices solve systems of differential equations to model and predict climate and weather patterns

Lesson 3

Stacks

Stacks

Lesson 3

Sets

What is a stack?

Lesson 3

Stacks

The stack is an analogy to a set of physical items stacked one atop another, such as a stack of plates.

Lesson 3

Stacks



Stacks

Stacks are collections of elements, which support two main operations:

- ▶ **PUSH**: which adds an element to the collection
- ▶ **POP**: which removes the most recent elements from the collection

Stacks

There might be, another operation called **PEEK**, which without modifying the stack, return the value of the last element added.

Stacks

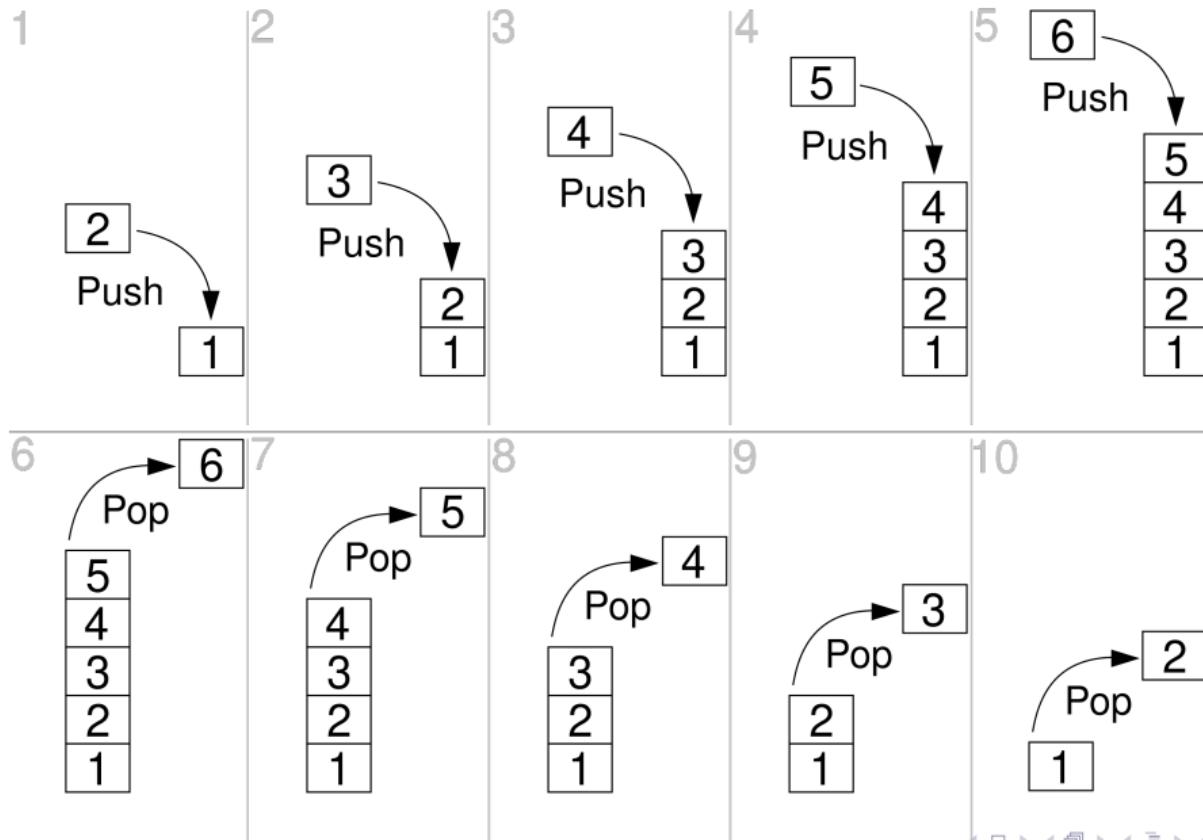
- ▶ **PUSH**: which adds an element to the collection
- ▶ **POP**: which removes the most recent elements from the collection
- ▶ **PEEK**: returns the value of the last element added

Stacks

The stack supports few operations. And it operates in a certain, predefined order. The order in which an element added to or removed from a stack is described as last in, first out, referred to by the acronym **LIFO**

Lesson 3

Stacks



Stacks

"Stacks entered the computer science literature in 1946, when **Alan Turing** used the terms "bury" and "unbury" as a means of calling and returning from subroutines."

Stacks

How can we implement a stack?

Stacks

A stack can be easily implemented using an array. The first element, usually at the zero offset, is the bottom, resulting in `array[0]` being the first element pushed onto the stack and the last element popped off.

Stacks

The program must keep track of the size (length) of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).

Stacks

The program must keep track of the size (length) of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).

Stack using arrays

```
# Create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack
```

```
# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0
```

```
# Add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print(item + "-pushed-to-stack-")
```

```
# Remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        # return minus infinite
        return str(-maxsize -1)
    return stack.pop()
```

```
# Return the top from stack without removing it
def peek(stack):
    if (isEmpty(stack)):
        # return minus infinite
        return str(-maxsize -1)
    return stack[len(stack) - 1]
```

Stack using LiFoQueue

```
from queue import LifoQueue  
stack = LifoQueue()  
stack.put(1)  
stack.put(2)  
stack.put(3)  
print(stack.get()) # Prints 3  
print(stack.get()) # Prints 2
```

There are numerous applications of stacks

- ▶ Undo mechanism in text editors
- ▶ Fwd and back buttons on web browsers
- ▶ Memory management in computer programming (static memory allocation. It can be used to keep track of functions calls)
- ▶ Implementing **recursion**

Lesson 3

Stacks

But what is recursion?

Recursion

- ▶ It is a programming technique, a way to program and develop a program where a function calls itself many times
- ▶ You take a big problem, into smaller problems, trying to apply a solution to solve the small problems
- ▶ You define a problem in terms of itself

Recursion

You are standing in a long queue of people. You must answer, how many people are behind you in the line?

Note: One person can see only the person standing directly in front and behind. One cannot look back and count. Each person is allowed to ask questions from the person standing in front or behind.

Recursion

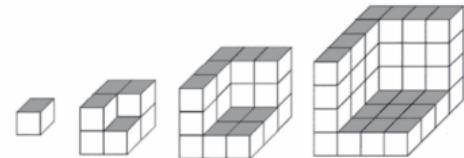
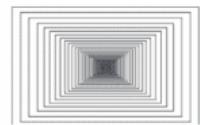
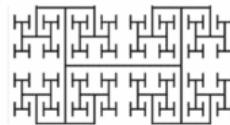
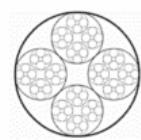
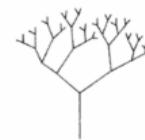
You look behind and see if there is a person there. If not, then you can return the answer "0". If there is a person, repeat this step and wait for a response from the person standing behind. Once a person receives a response, they add 1 and respond to the person that asked them or the person standing in front of them.

Recursion example

```
1: procedure PERSONCOUNT(currPerson)
2:   if noOneBehind(currPerson) == TRUE then
3:     return 0
4:   else personBehind == currPerson.checkBehind
5:     return 1 + PERSONCOUNT()
6:   end if
7: end procedure
```

Thinking recursively

- ▶ Learning to look for big things
- ▶ That are made from smaller things



Solving a problem recursively

Recursion is a function calling itself until a generic, base condition is true to produce the correct output. In other words, to solve a problem, we solve a problem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem.

Factorial number

In mathematics, the factorial of a non-negative integer $n!$ is the product of all positive integers less than or equal to n

$$n! = n \times (n - 1)! \quad (1)$$

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1 \quad (2)$$

Lesson 3

Stacks

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

Lesson 3

Stacks

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800

11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 000
16	20 922 789 888 000
17	355 687 428 096 000
18	6 402 373 705 728 000
19	121 645 100 408 832 000
20	2 432 902 008 176 640 000
25	$1.551\ 121\ 004 \times 10^{25}$
50	$3.041\ 409\ 320 \times 10^{64}$

Recursion example

```
1: procedure FACTORIAL( $N$ )
2:   if  $N == 0$  then
3:     return 1
4:   else
5:     return  $N * \text{FACTORIAL}(N - 1)$ 
6:   end if
7: end procedure
```

Factorial, using recursion

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

print( factorial (5))
```

Solving a problem recursively

For a recursive algorithm to work, smaller subproblems must be found and arrive at the base case. In simple words, any recursive algorithm has two parts: the base case and the recursive structure.

The Base Case

The base case is a terminating condition where a function immediately returns the result. This is the smallest version of the problem for which we already know the solution.

The Recursive structure

The recursive structure is an idea to design a solution to a problem via the solution of its smaller sub-problems, i.e., the same problem but for a smaller input size. We continue calling the same problem for smaller input sizes until we reach the base case of recursion.

How recursion works?

If we draw the flow of recursion for the factorial program, one can find this pattern: we are calling `fact(0)` last, but it is returning the value first. Similarly, we are calling `fact(n)` first, but it is returning the value last. Its a Last In First Out (LIFO) order for all recursive calls and return values.

Recursion uses Stacks behind

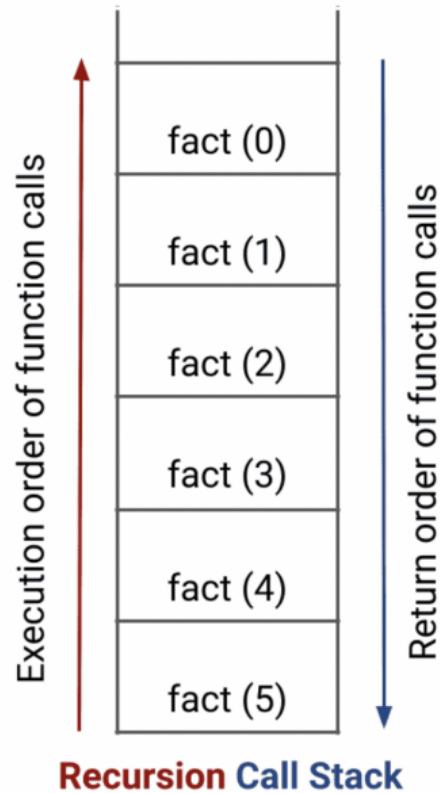
Order of recursive calls: larger problem to smaller problem
 $fact(n) \rightarrow fact(n - 1) \rightarrow \dots \rightarrow fact(1) \rightarrow fact(0)$

Order of return values: smaller problem to larger problem
 $fact(0) \rightarrow fact(1) \rightarrow \dots \rightarrow fact(n - 1) \rightarrow fact(n)$

Lesson 3

Stacks

Execution call stack!



Recursion is important!

- ▶ the basis for Dynamic Programming and Divide and Conquer algorithms
- ▶ helps in solving complex problems by breaking them into smaller subproblems
- ▶ fundamental to sorting, like quicksort, mergesort
- ▶ used in traversing trees and other complex data structures

Recursion vs Iteration?

A program is called **recursive** when an entity calls itself. A program is called **iterative** when there is a loop (or repetition) of some sort.

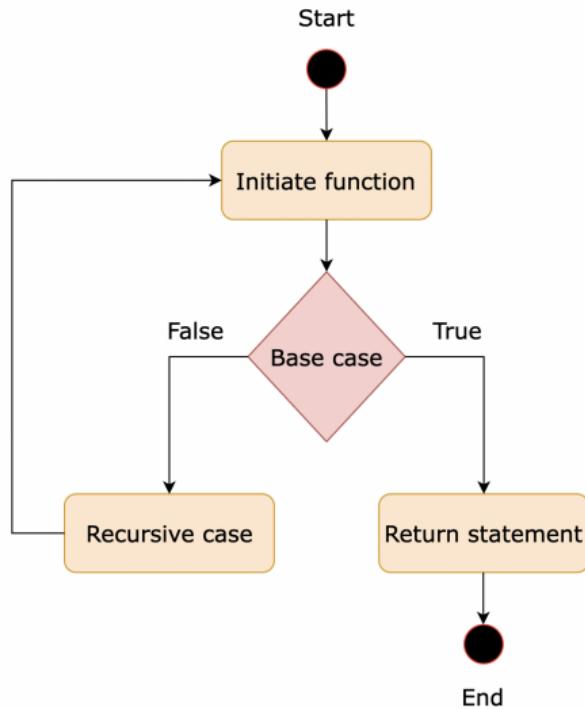
Recursion

- ▶ each function call creates a smaller problem to solve
- ▶ and these calls continue until reaching a basic case which is trivial to solve



Recursive function

- ▶ **Base case** The condition under which the function stops calling itself. This prevents infinite recursion and provides a direct answer for the simplest instance of the problem.
- ▶ **Recursive case** The part of the function that reduces the problem into smaller instances and calls itself with small data.



Recursive Factorial

```
def factorial (n):
    # Base case: if n is 1 or 0, factorial is 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: n * factorial of (n-1)
    else:
        return n * factorial (n - 1)
print( factorial (5))
```

Recursion Types

- ▶ direct recursion
- ▶ indirect recursion
- ▶ tail recursion

Direct Recursion - the simplest form of recursion. A function directly calls itself within its definition.

```
def direct_recursion (n):
    if n <= 0:
        return
    print(n)
    direct_recursion (n - 1)
```

Indirect Recursion - creates cycles of function calls. A function calls another function, which calls the original function.

```
def functionA(n):
    if n <= 0:
        return
    print(n)
    functionB(n - 1)

def functionB(n):
    if n <= 0:
        return
    print(n)
    functionA(n - 2)
```

Tail Recursion - the recursive call is the last operation in the function.

```
def tail_recursion (n, aggregate=1):
    if n == 0:
        return aggregate
    return tail_recursion (n - 1, n * aggregate)
```

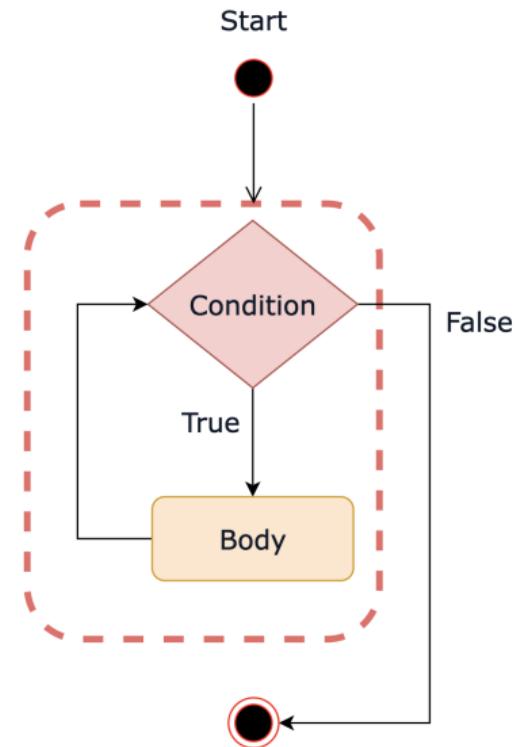
Iterative

- ▶ a repetitive process
- ▶ runs a number of times until a specified condition is met
- ▶ or certain number of iterations have been reached



Iterative function

- ▶ **Initialization** Start with initializing variables or setting initial conditions required for the iterative process
- ▶ **Condition** Check a condition that determines whether the iteration should continue or stop
- ▶ **Body** Execute a set of instructions or operations that represent the core logic of the iteration.



How to build an iterative function?

- ▶ loops: while, do-while, for loops
- ▶ for loop: when the number of iterations is known beforehand
- ▶ while loop: when the number of iterations is unknown, and the loop continues as long as a condition is true
- ▶ do-while loop: same as while loop but ensures that the loop is executed at least once before the condition is tested

Iterative Factorial - the iterative function to calculate the factorial

```
def factorial (n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print( factorial (5))
```

When to use recursion vs. iteration

The nature of the problem

- ▶ naturally fit a recursive structure or require breaking down into smaller subproblems
- ▶ for loop: when the number of iterations is known beforehand
- ▶ use iteration for problems with a straightforward repetitive process or when the number of iterations is known beforehand

When to use recursion vs. iteration

Complexity and readability

- ▶ Use recursion when it simplifies the code and makes it more readable
- ▶ Use iteration when recursion would make the code unnecessarily complex or when avoiding the risk of stack overflow is essential

When to use recursion vs. iteration

Performance consideration

- ▶ Use recursion if the problem's recursive nature allows for more elegant and maintainable code and manageable stack usage
- ▶ Use iteration if memory usage is a concern and the problem can be solved efficiently with loops.

Use recursion

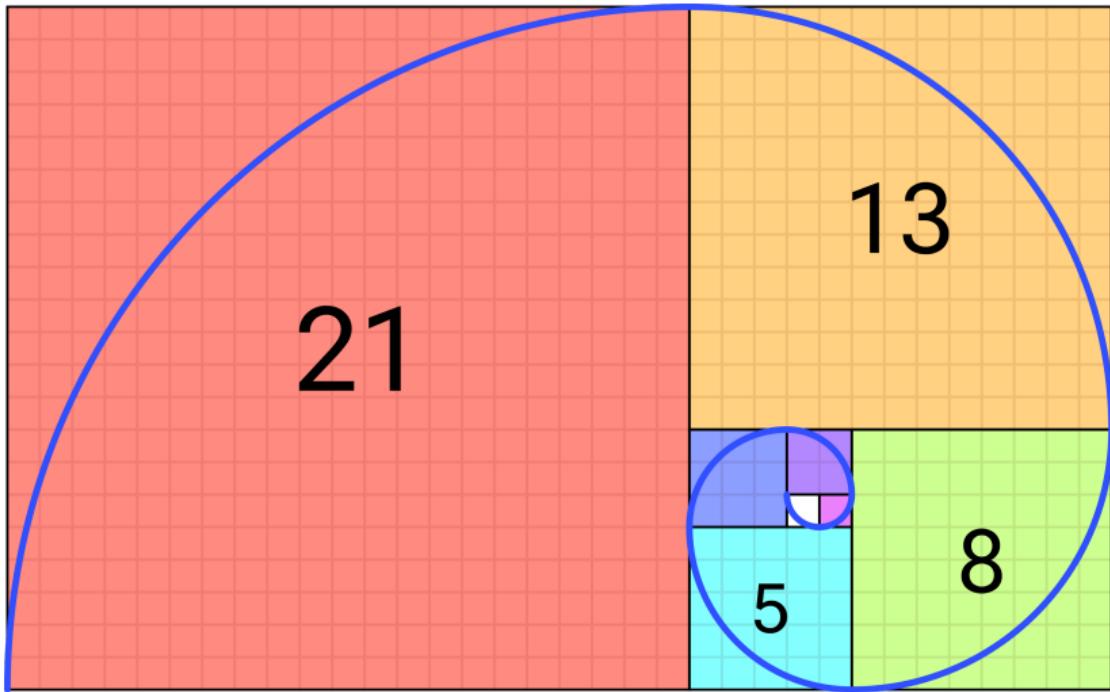
- ▶ **Divide and conquer** problems like merge sort, quicksort, and binary search where the problem is divided into smaller subproblems that are solved recursively
- ▶ **Tree and graph problems** Traversals (e.g., in-order, pre-order, post-order for trees) and pathfinding in graphs that naturally fit a recursive approach.
- ▶ **Dynamic programming** Problems like the Fibonacci sequence, where recursion with memoization simplifies the solution
- ▶ **Combinatorial problems** Generating permutations and combinations and solving puzzles like the Tower of Hanoi

Use iteration

- ▶ **Simple loops** Problems like summing a list of numbers, iterating through arrays or lists, and simple counting problems
- ▶ **Linear problems** Iterative solutions for problems requiring a linear scan, such as finding an array's maximum or minimum value
- ▶ **Repetitive tasks** Problems that require repeating a task a known number of times, such as printing numbers from 1 to N or iterating through data structures like arrays and linked lists.

Lesson 3

Stacks



Fibonacci sequence

Fibonacci sequence

In mathematics, the **Fibonacci sequence** is a [sequence](#) in which each element is the sum of the two elements that precede it. Numbers that are part of the Fibonacci sequence are known as **Fibonacci numbers**, commonly denoted F_n . Many writers begin the sequence with 0 and 1, although some authors start it from 1 and 1^{[1][2]} and some (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the sequence begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... (sequence [A000045](#) in the [OEIS](#))

Lesson 3

Stacks

Fibonacci sequence

Definition [edit]

The Fibonacci numbers may be defined by the [recurrence relation](#)^[7]

$$F_0 = 0, \quad F_1 = 1,$$

and

$$F_n = F_{n-1} + F_{n-2}$$

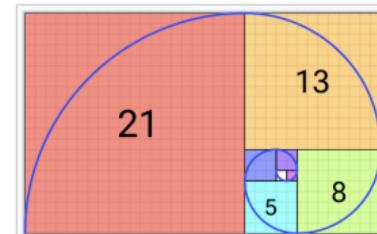
for $n > 1$.

Under some older definitions, the value $F_0 = 0$ is omitted, so that the sequence starts with $F_1 = F_2 = 1$, and the recurrence

$$F_n = F_{n-1} + F_{n-2} \text{ is valid for } n > 2.$$
^{[8][9]}

The first 20 Fibonacci numbers F_n are:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181



The Fibonacci spiral: an approximation of the [golden spiral](#) created by drawing [circular arcs](#) connecting the opposite corners of squares in the Fibonacci tiling (see preceding image)

Fibonacci Recursion

```
def fibonacci_recursive (n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive (n-1) + fibonacci_recursive (n-2)
```

Main Code

n = 10

```
print(f"Recursive-method:-Fibonacci({{n}})={{ fibonacci_recursive (n)}}")
```

Fibonacci Iterative

```
def fibonacci_iterative (n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Main code

n = 10

```
print(f" Iterative -method:-Fibonacci({{n}}) -={{ fibonacci_iterative (n)}}")
```

Advantages of Stacks

- ▶ Simplicity: Stacks are a simple and easy-to-understand data structure
- ▶ Efficiency: Push and pop operations on a stack are very fast, providing efficient access to data.
- ▶ LIFO: Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed.
- ▶ Limited memory usage: Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

Disadvantages of Stacks

- ▶ Limited access: Elements can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- ▶ Overflow: If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- ▶ No random access: Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- ▶ Limited capacity: Fixed capacity, a limitation if the number of elements that need to be stored is unknown or highly variable.

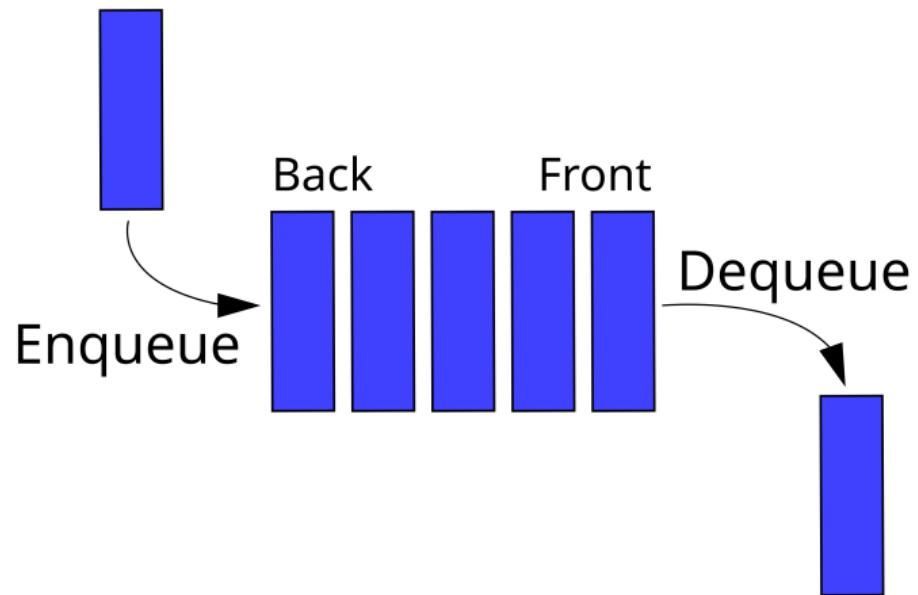
Queues

Queues

A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.

Lesson 3

Queues



By convention, the end of the sequence at which elements are added is called **the back, tail, or rear** of the queue, and the end at which elements are removed is called the **head or front** of the queue, analogously to the words used when people line up to wait for goods or services.

Queues

The operations of a queue make it a first-in-first-out **FIFO** data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queue Properties

- ▶ **Front:** Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (head of the queue)
- ▶ **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the rear of the queue. (tail of the queue)
- ▶ **Size:** the current number of elements in the queue
- ▶ **Capacity:** the maximum number of elements the queue can hold

Queue Operations

- ▶ **Enqueue** — Add an element to the end of the queue
- ▶ **Dequeue** — Remove an element from the front of the queue
- ▶ **Is Empty** — Check if the queue is empty
- ▶ **Is Full** — Check if the queue is full
- ▶ **Peek** — Get the value of the front element without removing it

Queue implements a basic FIFO queue

```
from queue import Queue  
q = Queue()  
q.put(1) # Add 1 to queue  
q.put(2)  
q.put(3)  
print(q.qsize()) # Prints 3  
print(q.get()) # Prints 1  
print(q.get()) # Prints 2
```

Python Queue

```
from queue import Queue  
q = Queue()  
# The key methods available are:  
# qsize() – Get the size of the queue  
# empty() – Check if queue is empty  
# full() – Check if queue is full  
# put(item) – Put an item into the queue  
# get() – Remove and return an item from the queue  
# join() – Block until all tasks are processed
```

Lesson 3

Linked lists

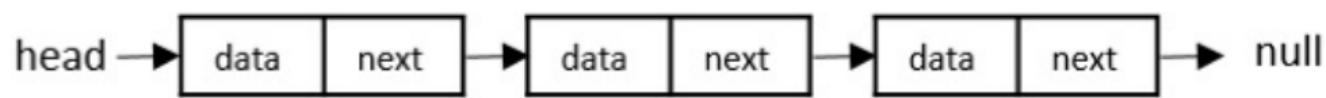
Linked lists

Linked Lists

A linked list is a linear collection of data elements where each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

Lesson 3

Linked lists

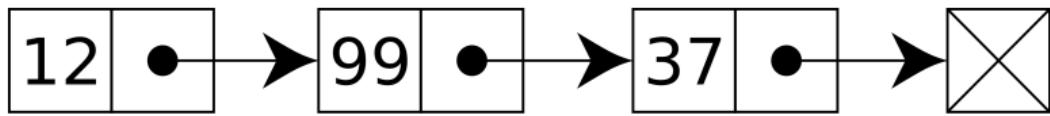


Linked Lists

Each node stores both data and a **reference (a pointer)** to the next node in the sequence.

Lesson 3

Linked lists



Linked Lists

Each node contains data, and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

Linked Lists

A drawback of linked lists is that data access time is linear in respect to the number of nodes in the list. Because nodes are serially linked, accessing any node requires that the prior node be accessed beforehand.

Linked Lists

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including stacks, queues, **associative and dynamic arrays**.

Dynamic Arrays

Variable-size list data structure that allows elements to be added or removed. Dynamic arrays overcome a limit of static arrays, which have a fixed capacity that needs to be specified at allocation.

Associative Arrays

A map, symbol table, or dictionary is an abstract data type that stores a collection of (key, value) pairs, such that each possible key appears at most once in the collection. It supports lookup, remove, and insert operations.

Associative Arrays

0		110
1		100
2		90
3		80
4		70
5		50

Linked lists vs. Arrays

Linked lists vs. Arrays

Linked lists

- ▶ Dynamic in size
- ▶ Can contain any number of nodes
- ▶ Store various data types

Arrays

- ▶ Fixed in size
- ▶ Size is given at the time of creation
- ▶ Similar data types

Linked Lists

The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items do not need to be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation.

Linked Lists

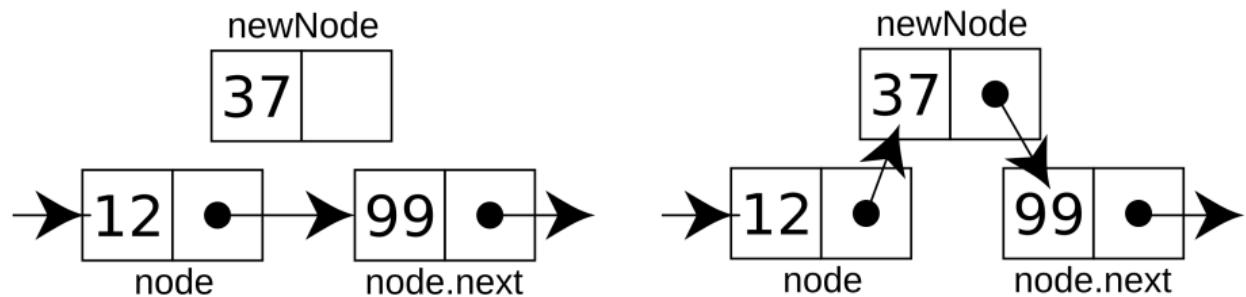
Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

Linked list Operations

- ▶ **INSERT** — Add an element to the end of the queue
- ▶ **DELETE** — Remove an element from the front of the queue
- ▶ **TRAVERSAL** — Check if the queue is empty

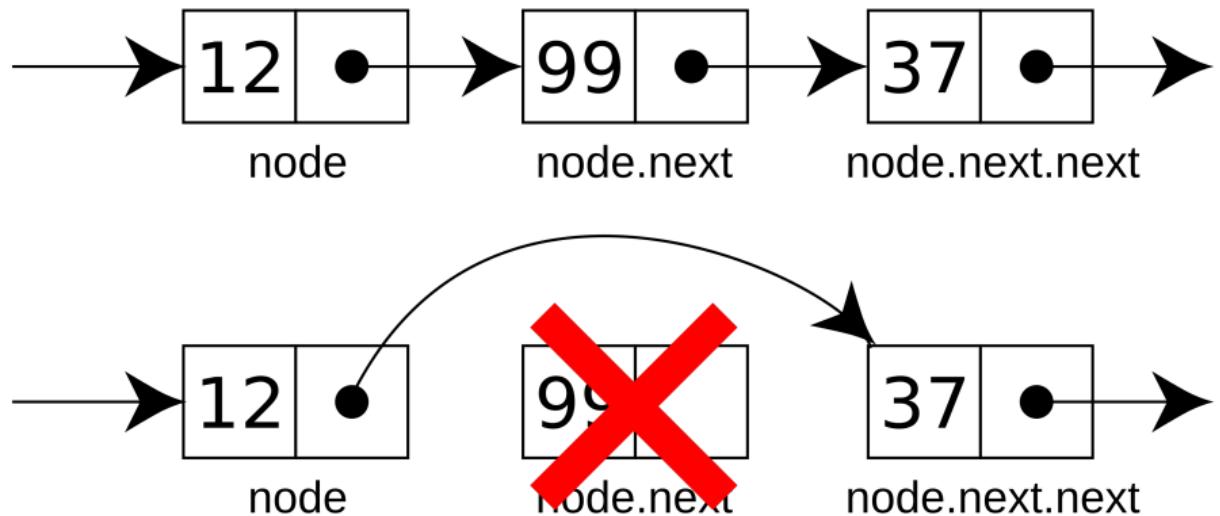
Lesson 3

Linked lists



Lesson 3

Linked lists



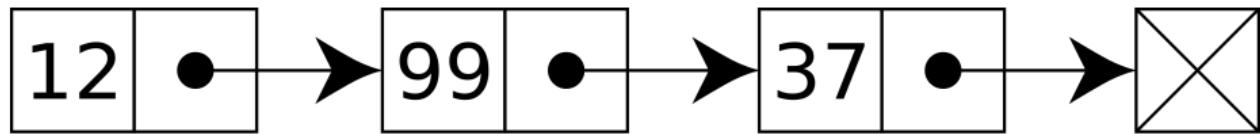
Linked lists

- ▶ **Single linked list** lists contain nodes which have a value and a next pointer
- ▶ **Double linked list** contains nodes which includes the next and previous pointer links
- ▶ **circular linked list** a list where the last node has a pointer to the first node

Lesson 3

Linked lists

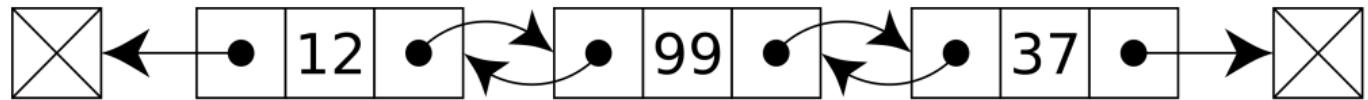
Single Linked list



Lesson 3

Linked lists

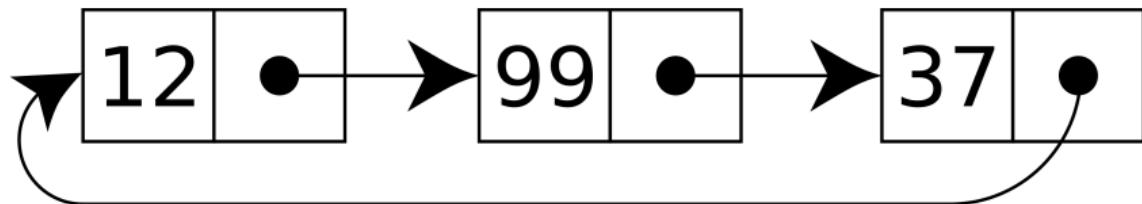
Double Linked list



Lesson 3

Linked lists

Circular Linked list



Linked List Example

```
# importing module
import collections

# initialising a deque() of arbitrary length
linked_list = collections.deque()
# filling deque() with elements
linked_list.append('subaru')
linked_list.append('toyota')
linked_list.append('mercedes')

print("Elements-in-the- linked_list :")
print( linked_list )
```

Hash Table

Linked Lists

A hash table is a data structure that implements an associative array, also called a dictionary or simply map; an associative array is an abstract data type that maps keys to values.

Remember Associative Arrays

A map, symbol table, or dictionary is an abstract data type that stores a collection of (key, value) pairs, such that each possible key appears at most once in the collection. It supports lookup, remove, and insert operations.

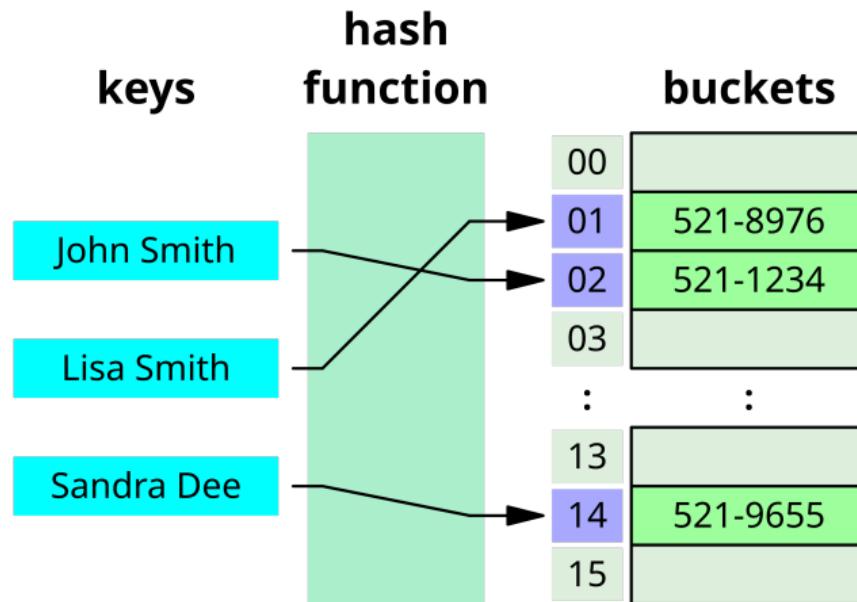
Hash Table

A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored. A map implemented by a hash table is called a hash map.

Lesson 3

Hash Table

Hash Table



Hash Table

Hashing concept - each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.

Hash Functions: cyclic redundancy checks

Name	Length	Type
cksum (Unix)	32 bits	CRC with length appended
CRC-8	8 bits	CRC
CRC-16	16 bits	CRC
CRC-32	32 bits	CRC
CRC-64	64 bits	CRC

Lesson 3

Hash Table

Hash Functions: checksums

Name	Length	Type
BSD checksum (Unix)	16 bits	sum with circular rotation
SYSV checksum (Unix)	16 bits	sum with circular rotation
sum8	8 bits	sum
Internet Checksum	16 bits	sum (ones' complement)
sum24	24 bits	sum
sum32	32 bits	sum
fletcher-4	4 bits	sum
fletcher-8	8 bits	sum
fletcher-16	16 bits	sum
fletcher-32	32 bits	sum
Adler-32	32 bits	sum
xor8	8 bits	sum
Luhn algorithm	1 decimal digit	sum
Verhoeff algorithm	1 decimal digit	sum
Damm algorithm	1 decimal digit	Quasigroup operation

Hash Functions: universal

Name	Length	Type [hide]
Rabin fingerprint	variable	multiply
tabulation hashing	variable	XOR
universal one-way hash function		
Zobrist hashing	variable	XOR

Lesson 3

Hash Table

Hash Functions: keyed cryptographic

Name	Tag Length	Type
BLAKE2		keyed hash function (prefix-MAC)
BLAKE3	256 bits	keyed hash function (supplied IV)
HMAC		
KMAC	arbitrary	based on Keccak
MD6	512 bits	Merkle tree NLFSR
One-key MAC (OMAC; CMAC)		
PMAC (cryptography)		
Poly1305-AES	128 bits	nonce-based
SipHash	32, 64 or 128 bits	non-collision-resistant PRF
HighwayHash^[16]	64, 128 or 256 bits	non-collision-resistant PRF
UMAC		
VMAC		

Lesson 3

Hash Table

Hash Functions: unkeyed cryptographic

HAVAL	128 to 256 bits	hash
JH	224 to 512 bits	hash
LSH ^[19]	256 to 512 bits	wide-pipe Merkle–Damgård construction
MD2	128 bits	hash
MD4	128 bits	hash
MD5	128 bits	Merkle–Damgård construction
MD6	up to 512 bits	Merkle tree NLFSR (it is also a keyed hash function)
RadioGatún	arbitrary	ideal mangling function
RIPEMD	128 bits	hash
RIPEMD-128	128 bits	hash
RIPEMD-160	160 bits	hash
RIPEMD-256	256 bits	hash
RIPEMD-320	320 bits	hash
SHA-1	160 bits	Merkle–Damgård construction
SHA-224	224 bits	Merkle–Damgård construction
SHA-256	256 bits	Merkle–Damgård construction
SHA-384	384 bits	Merkle–Damgård construction
SHA-512	512 bits	Merkle–Damgård construction
SHA-3 (subset of Keccak)	arbitrary	sponge function

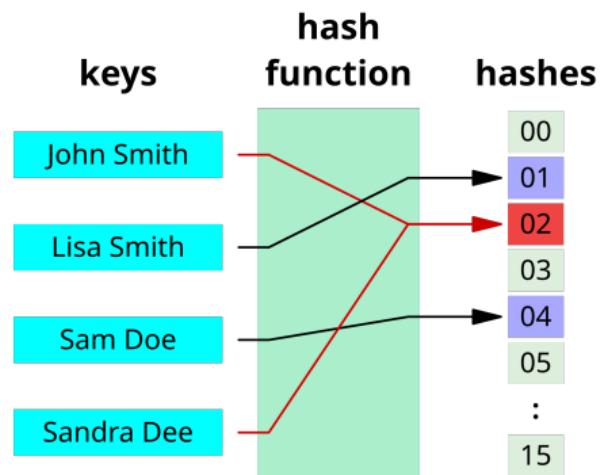
Hash Collisions

A hash collision or hash clash is when two distinct pieces of data in a hash table share the same hash value. The hash value in this case is derived from a hash function which takes a data input and returns a fixed length of bits.

Lesson 3

Hash Table

Hash Collisions



Hash Table

Hash tables are very much used as table lookup structures, in many kinds of computer software, particularly database indexing, caches, and sets.

Hash Table

In hash tables, since hash collisions are inevitable, hash tables have mechanisms of dealing with them, known as collision resolutions.

- ▶ **open addressing** cells in the hash table are assigned one of three states in this method – occupied, empty, or deleted
- ▶ **separate chaining** allows more than one record to be chained to the cells of a hash table.

Hash Table

```
hash_table = {"Alice": "January", "Bob": "May", "Charlie": "January"}  
# Hash function determines the location for "January"
```

```
my_dictionary = {}  
my_dictionary["Alice"] = "January"  
# Hash function determines the location for "January"
```

```
# print the key value
```

```
print( my_dictionary["Alice"] ) # "January"
```

Lesson 3

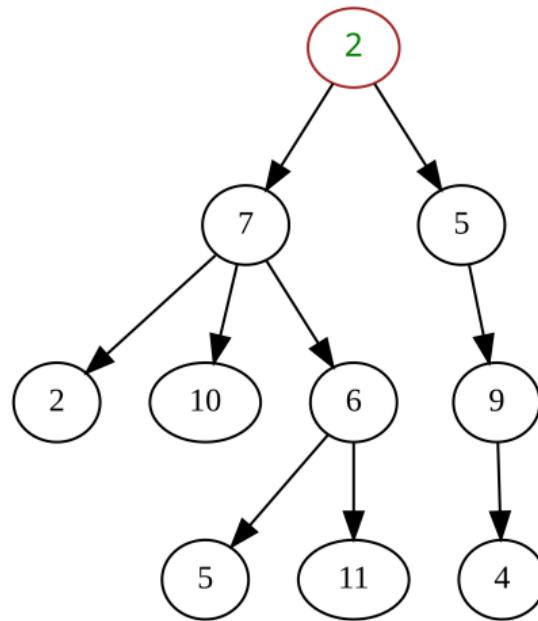
Trees

Trees

What is a tree?

A tree is a very widely used abstract data type that represents a hierarchical structure with a set of connected nodes. Each node in the tree can be connected to many children (depending on the type of tree), but must be connected to exactly one parent, except for the root node, which has no parent

Unsorted tree



Tree

There should be no cycles or "loops" (no node can be its own ancestor), and also that each child can be treated like the root node of its own subtree, making recursion a useful technique for tree traversal.

Tree Node

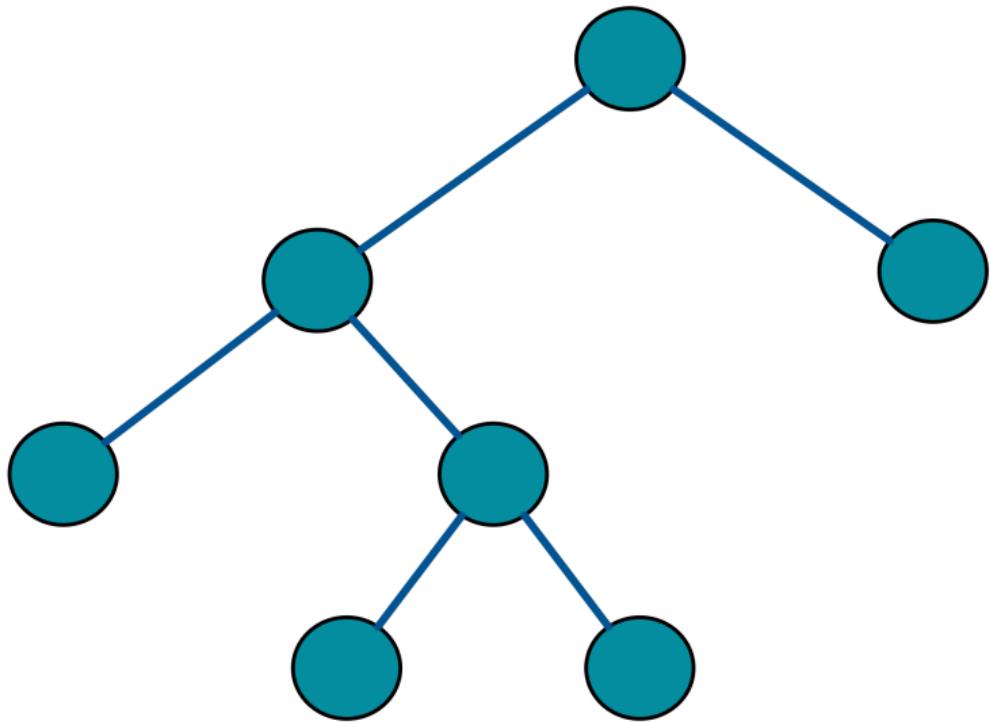
A node is a structure which may contain data and connections to other nodes, sometimes called edges or links. Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees are drawn with descendants going downwards).

Tree Node

A node that has a child is called the child's parent node (or superior). All nodes have exactly one parent, except the topmost root node, which has none. A node might have many ancestor nodes, such as the parent's parent.

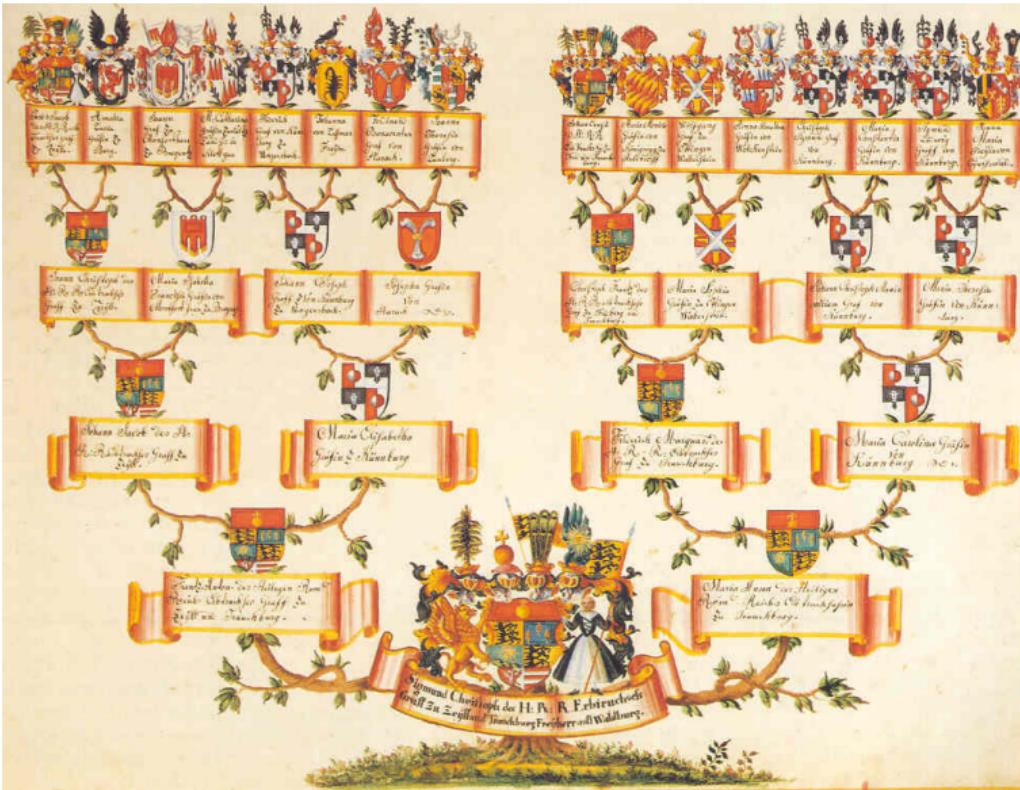
Lesson 3

Trees



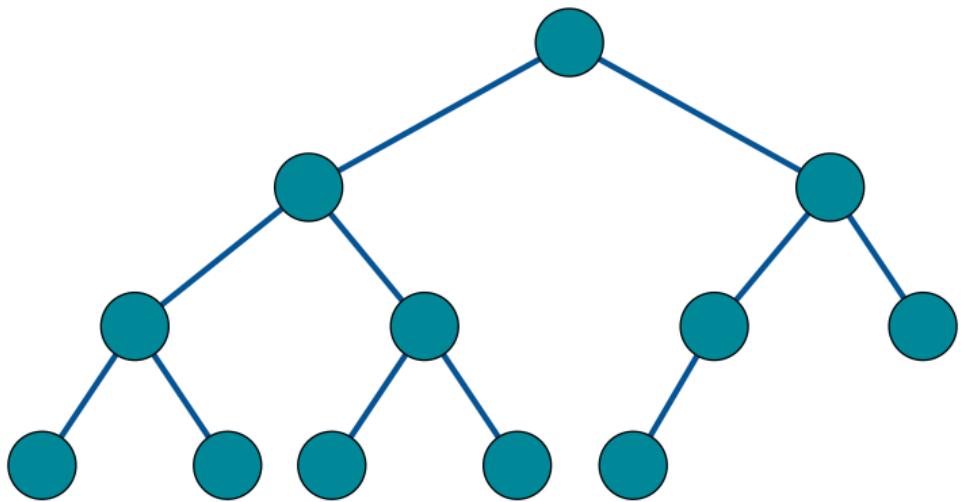
Lesson 3

Trees



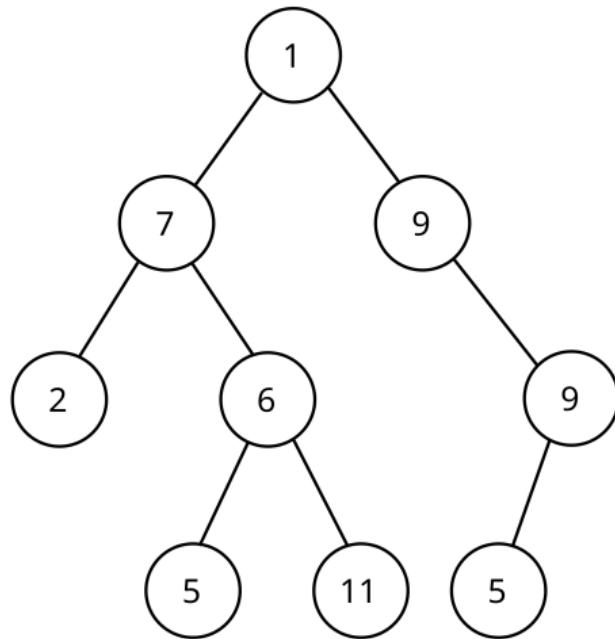
Lesson 3

Trees



Lesson 3

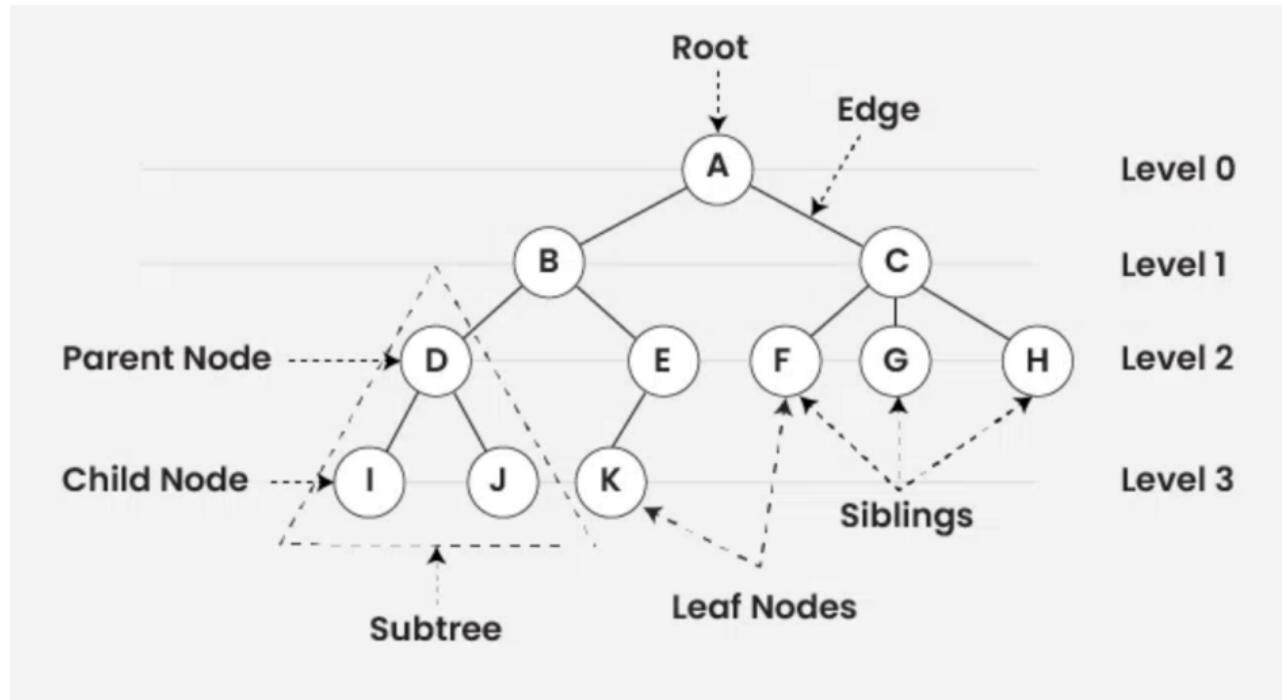
Trees



Lesson 3

Trees

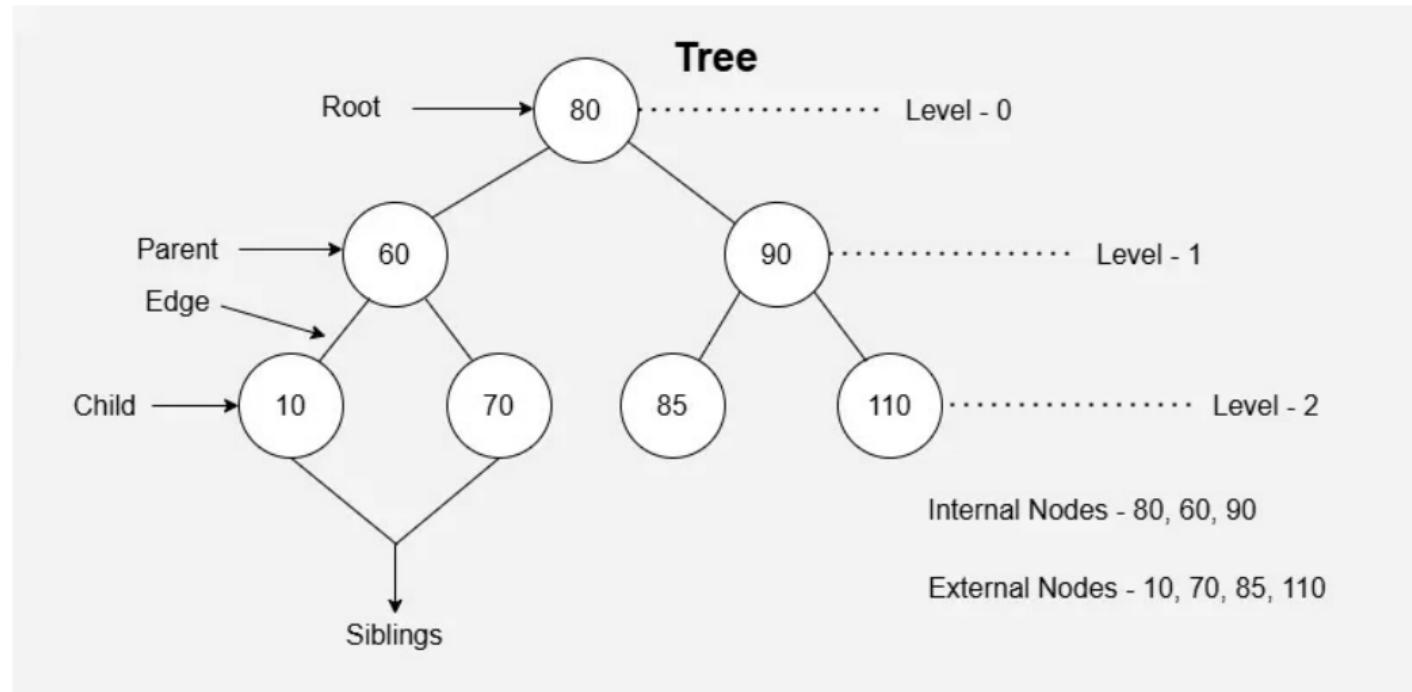
Structure



Lesson 3

Trees

Organization



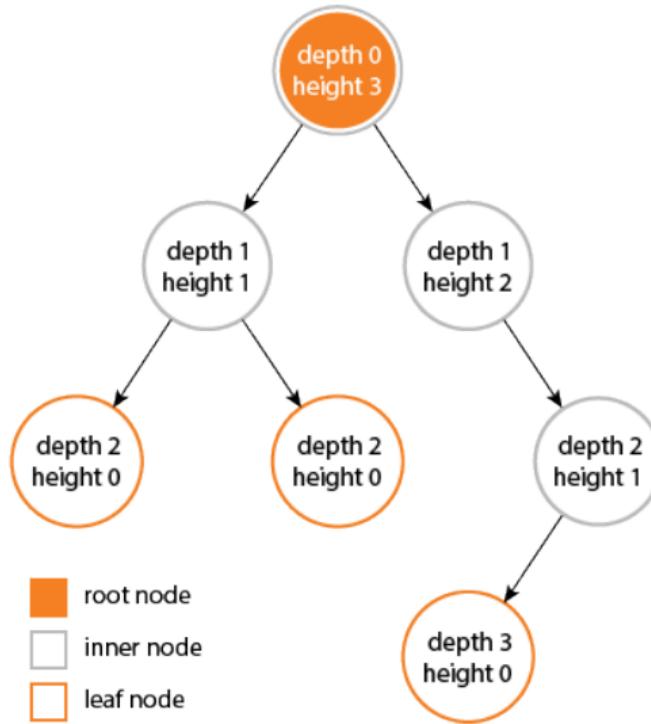
Tree - Properties

- ▶ **Number of edges** as the connection between two nodes. If a tree has N nodes then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree
- ▶ **Depth of a node** the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

Tree Properties

- ▶ **Height of a node** as the length of the longest path from the node to a leaf node of the tree.
- ▶ **Height of the Tree** is the length of the longest path from the root of the tree to a leaf node of the tree
- ▶ **Degree of a Node** the total count of subtrees attached to that node

Height vs. Depth



Tree - Non-linear data structure

- ▶ **not linear** data in a tree are not stored in a sequential manner
- ▶ **hierarchical structure** data arranged on multiple levels

Types of Trees

- ▶ **Binary tree**
- ▶ **Ternary Tree**
- ▶ **Generic Tree**

Lesson 3

Trees

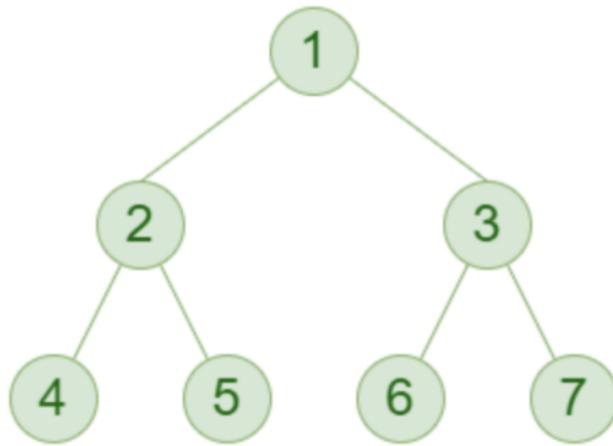
Binary Trees

Binary Tree

A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.

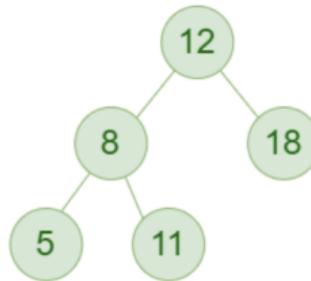
Lesson 3

Trees



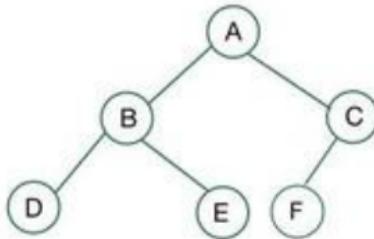
Full Binary Tree

A full binary tree is a binary tree with either zero or two child nodes for each node.



Complete Binary Tree

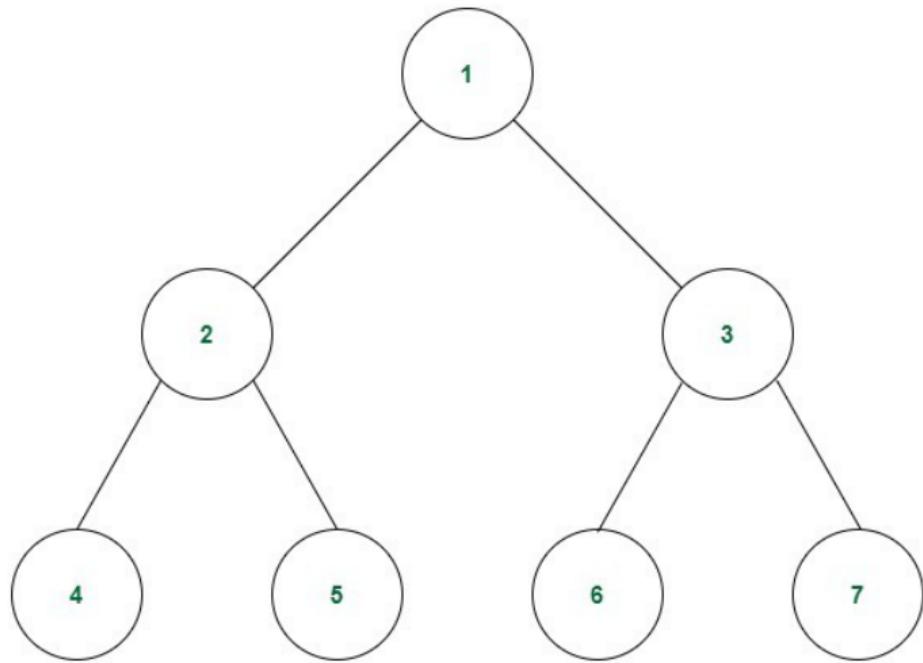
A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from left as possible.



Perfect Binary Tree

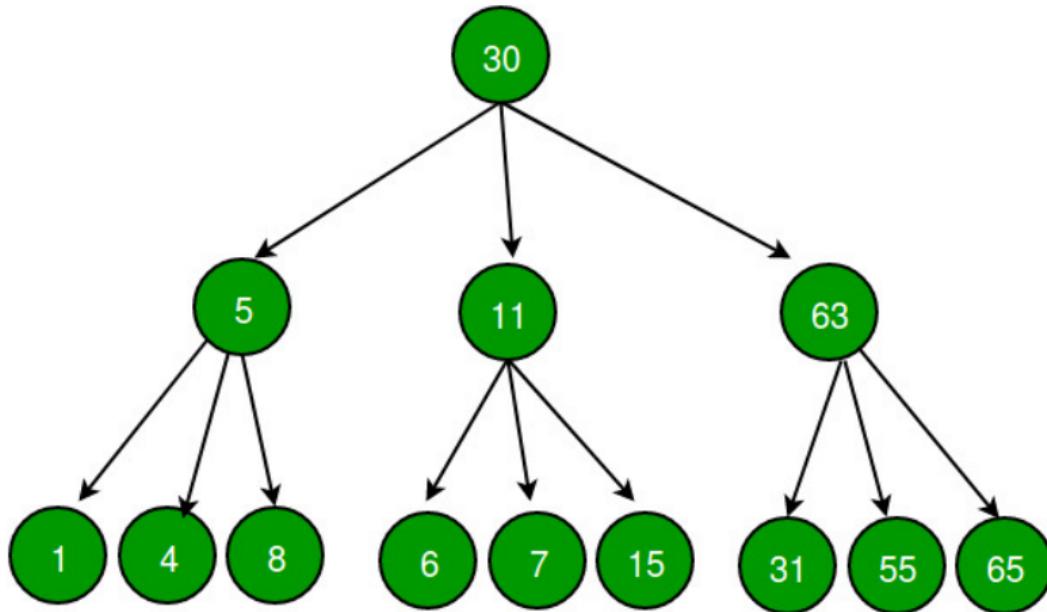
A perfect binary tree is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.

Perfect Binary Tree



Ternary Trees

Ternary Tree

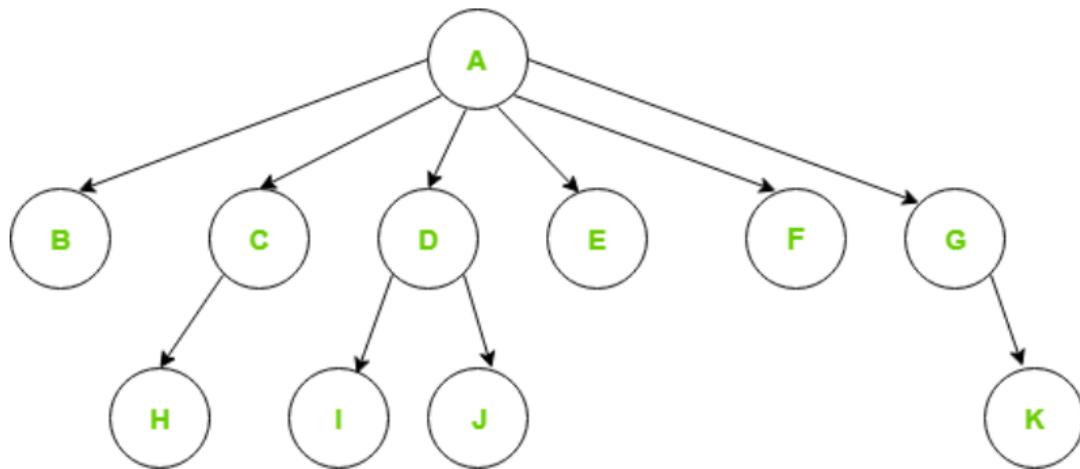


Lesson 3

Trees

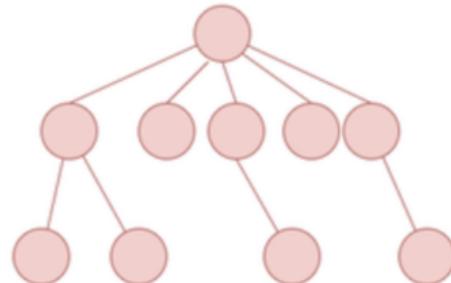
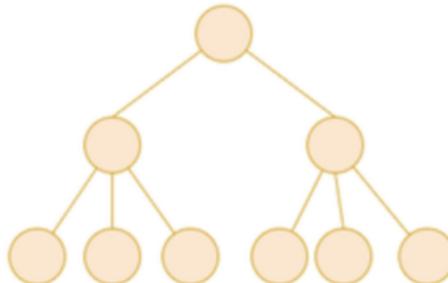
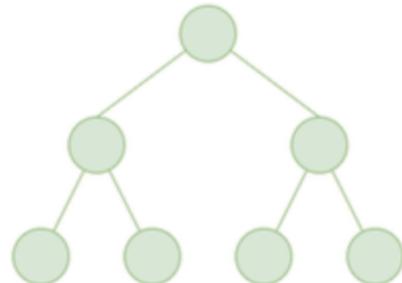
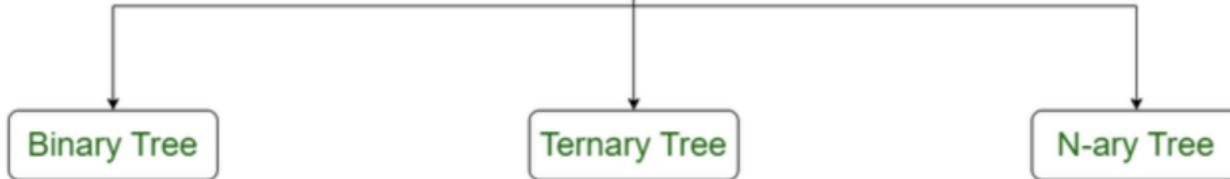
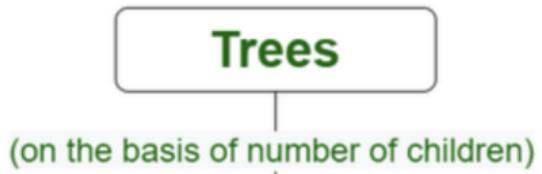
Generic Trees

Generic Tree



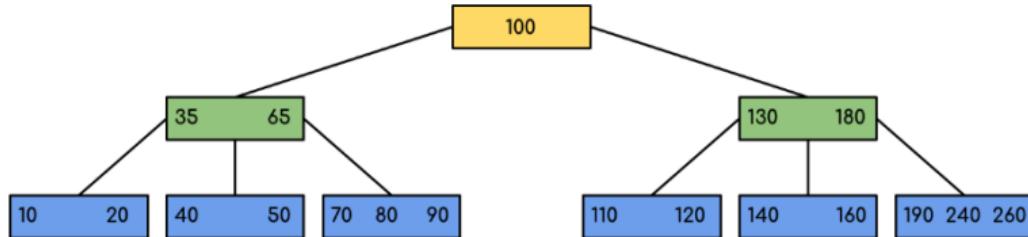
Lesson 3

Trees



Binary search tree (BST)

A B-Tree is a specialized N-way tree designed to optimize data access, especially on disk-based storage systems.



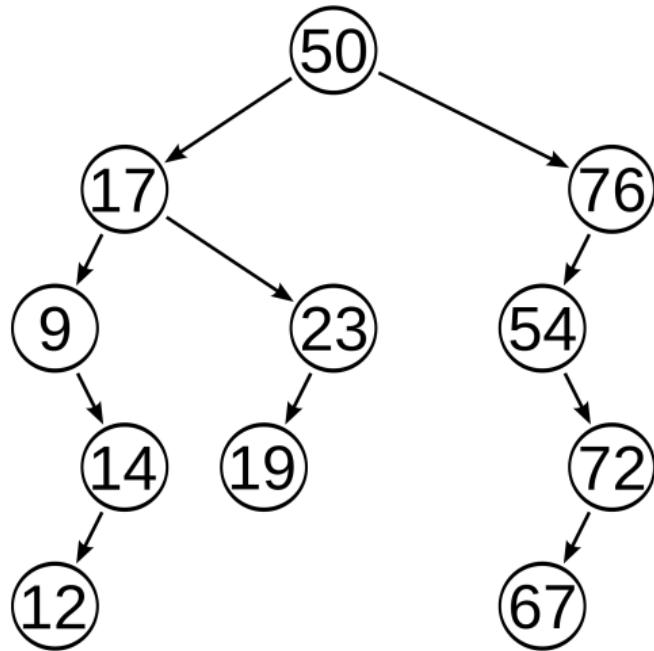
Binary Search Trees (BST)

A B-tree is a **self-balancing** tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions very fast (logarithmic time). The B-tree generalizes the binary search tree, allowing for nodes with more than two children.

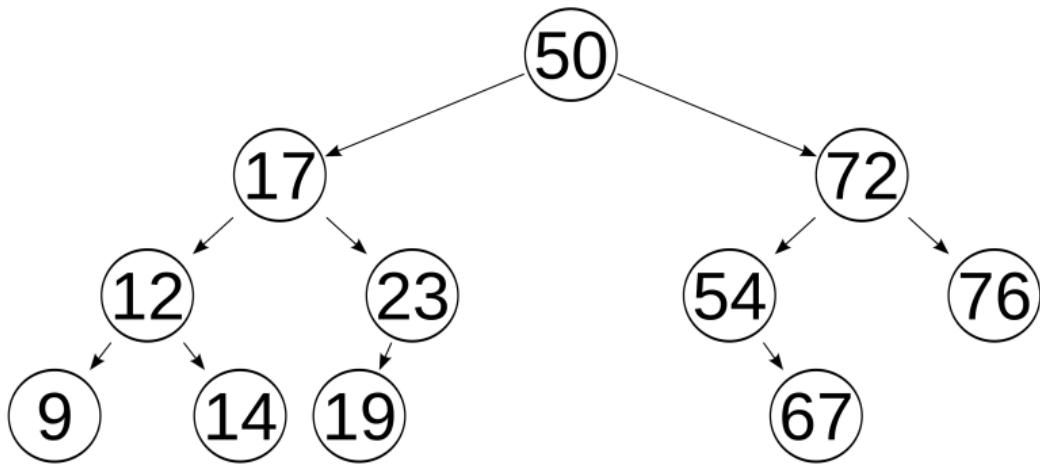
Self-Balancing?

A self-balancing tree is a tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.

Un-balanced tree



Self-balanced tree



Binary Search Trees (BST)

Binary search trees allow very fast lookup, addition, and removal of data items using binary search. Binary search trees are the basis and very important for sorting and search algorithms.

Tree Operations

- ▶ **CREATE** — create a tree in a data structure
- ▶ **INSERT** — insert data in a tree
- ▶ **SEARCH** — search data in a tree
- ▶ **TRAVERSAL** — Depth-First, Breadth-First

There are numerous applications of trees

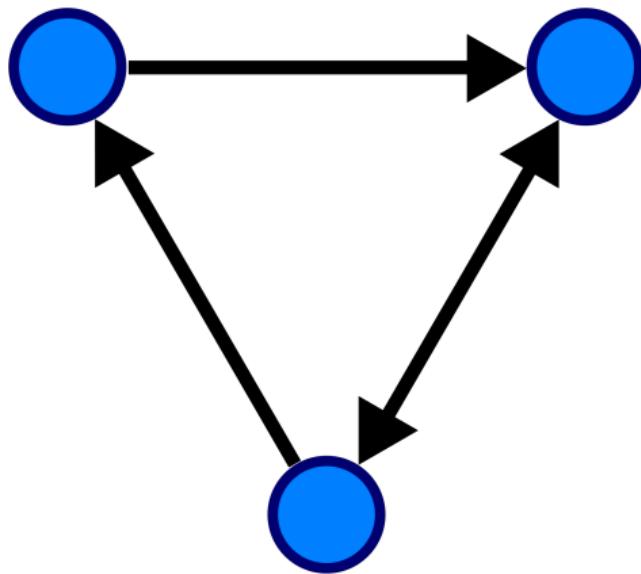
- ▶ File system implementation: directory structure
- ▶ Natural language processing (NLP)
- ▶ AI, Genetic programming
- ▶ Database indexing (BST B-Trees, B+ Trees)

Graphs

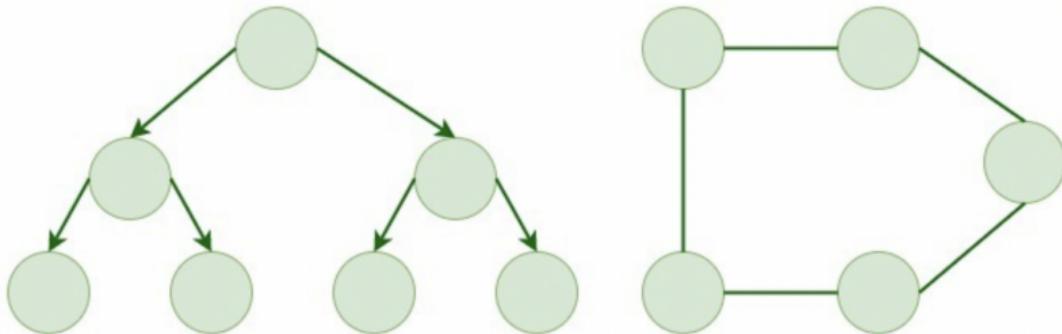
What is a graph?

A graph is a non-linear data structure consisting of vertices and edges. It is an abstract data type that consists of a finite set of vertices (nodes or points), together with a set of edges (also called links or lines).

Graphs



Graphs vs Trees



Graphs

A graph data structure is a collection of nodes (also called vertices) and edges that connect them. Nodes can represent entities, such as people, places, or things, while edges represent relationships between those entities.

Trees

A tree data structure is a hierarchical data structure that consists of nodes connected by edges. Each node can have multiple child nodes, but only one parent node. The topmost node in the tree is called the root node.

Lesson 3

Graphs

Feature	Graph	Tree
Definition	A collection of nodes (vertices) and edges, where edges connect nodes.	A hierarchical data structure consisting of nodes connected by edges with a single root node.
Structure	Can have cycles (loops) and disconnected components.	No cycles; connected structure with exactly one path between any two nodes.
Root Node	No root node; nodes may have multiple parents or no parents at all.	Has a designated root node that has no parent.
Node Relationship	Relationships between nodes are arbitrary.	Parent-child relationship; each node (except the root) has exactly one parent.
Edges	Each node can have any number of edges.	If there is n nodes then there would be $n-1$ number of edges
Traversal Complexity	Traversal can be complex due to cycles and disconnected components.	Traversal is straightforward and can be done in linear time.

Graphs vs Trees

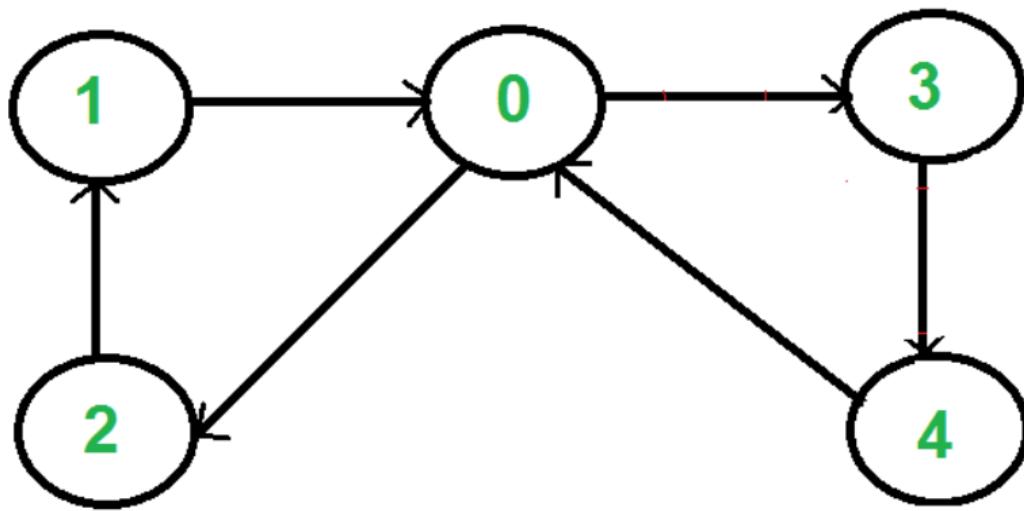
- ▶ **Cycles** Graphs can contain cycles, while trees cannot
- ▶ **Connectivity** Graphs can be disconnected, while trees are always connected
- ▶ **Hierarchy** Trees have a hierarchical structure, with one vertex designated as the root. Graphs do not have this hierarchical structure
- ▶ **Applications** Trees used in hierarchical data structures, such as file systems and XML documents. Graphs better for modelling transportation networks, social networks, computer networks

Types of graphs

- ▶ Directed and Undirected graphs
- ▶ Weighted Vs Unweighted graphs

Directed graph

A directed graph is defined as a type of graph where the edges have a direction associated with them.



Directed graphs

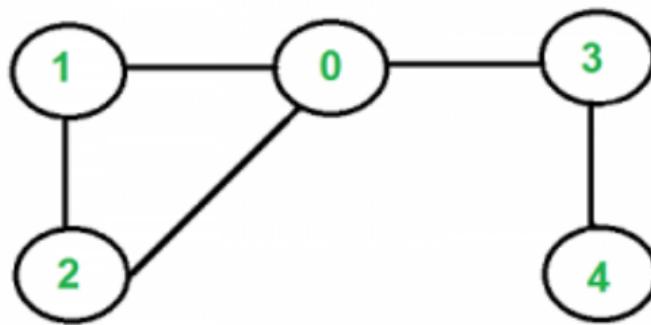
- ▶ edges have a direction associated with them, indicating a one-way relationship between vertices
- ▶ each vertex in a directed graph has two different degree measures: indegree and outdegree. Indegree is the number of incoming edges to a vertex, while outdegree is the number of outgoing edges from a vertex
- ▶ A directed graph can contain cycles, which are paths that start and end at the same vertex and contain at least one edge

There are numerous applications of directed graphs

- ▶ social networks
- ▶ transportation networks: flight routes, railroads, and road networks. Cities, towns, and intersections are represented by nodes, and the links that connect these locations—such as highways, train tracks, and flight routes—are represented by edges.
- ▶ computer networks

Undirected graph

An undirected graph is a type of graph where the edges have no specified direction assigned to them.



Undirected graphs

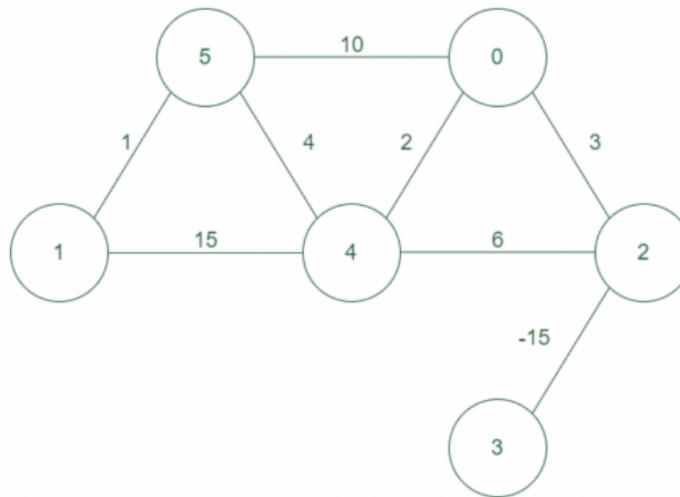
- ▶ edges in an undirected graph are bidirectional
- ▶ there is no concept of a “parent” or “child” vertex as there is no direction to the edges.
- ▶ may contain loops, which are edges that connect a vertex to itself

Applications of un-directed graphs

- ▶ traffic flow optimization: Undirected graphs are used in traffic flow optimization to model the flow of vehicles on road networks. The vertices of the graph represent intersections or road segments, and the edges represent the connections between them. The graph can be used to optimize traffic flow and plan transportation infrastructure
- ▶ website analysis: analyze the links between web pages on the internet

Weighted graph

A weighted graph is defined as a special type of graph in which the edges are assigned some weights which represent cost, distance, and many other relative measuring units.

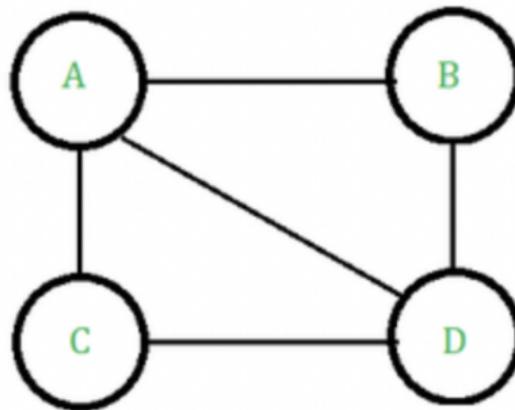


Applications of weighted graphs

- ▶ Transportation networks: solve the path that takes the least time, or the path with the least overall distance. This is a simplification of how weighted graphs can be used for more complex things like a GPS system. Graphs are used to study traffic patterns, traffic light timings and much more by many big tech companies such as UBER etc. Graph networks are used by many map programs such as Google Maps, Bing Maps
- ▶ Epidemiology: Weighted graphs can be used to find the maximum distance transmission from an infectious to a healthy person

Unweighted graph

An unweighted graph is a graph in which the edges do not have weights or costs associated with them. Instead, they simply represent the presence of a connection between two vertices.

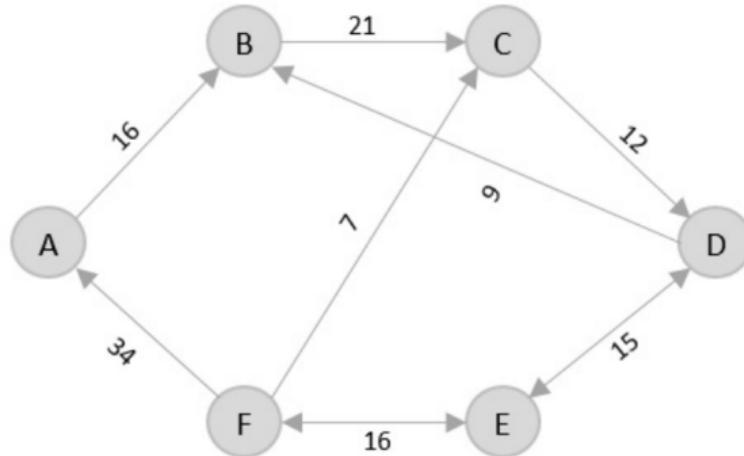


Applications of unweighted graphs

- ▶ represent circuit diagrams
- ▶ solve puzzles
- ▶ used in social media sites to find whether two users are connected or not
- ▶ can modelate the possible moves in a game

A real problem

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?



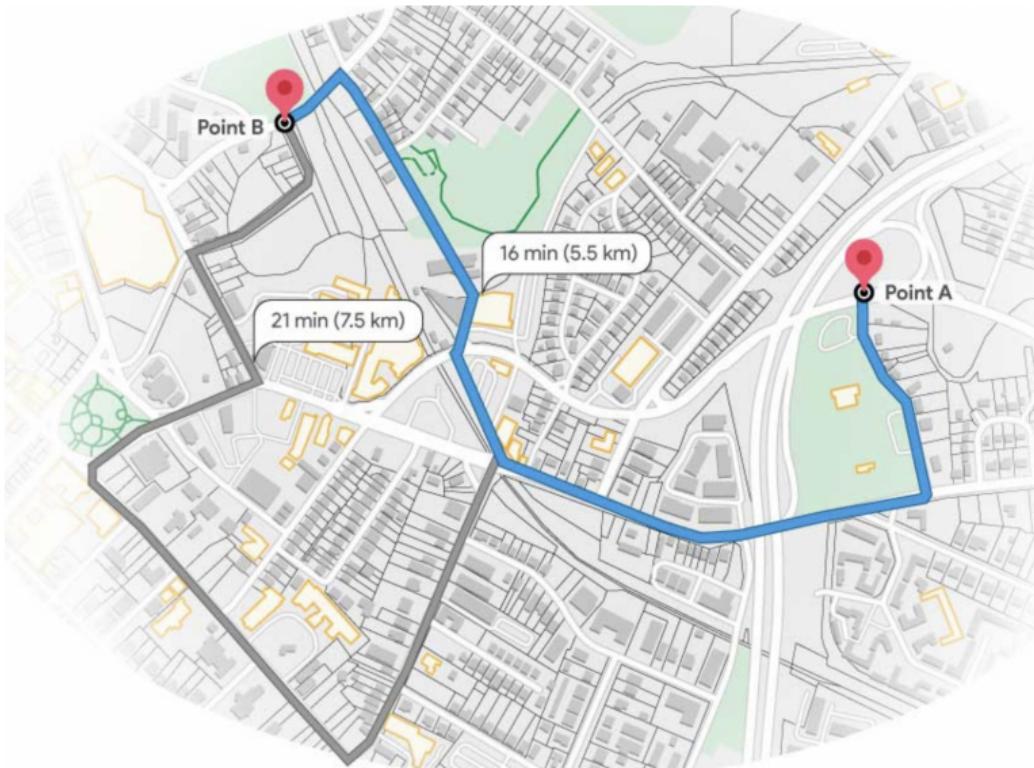
How we plan to solve the problem?

- ▶ think how we represent the problem
- ▶ what data structure we plan to use
- ▶ and what algorithm

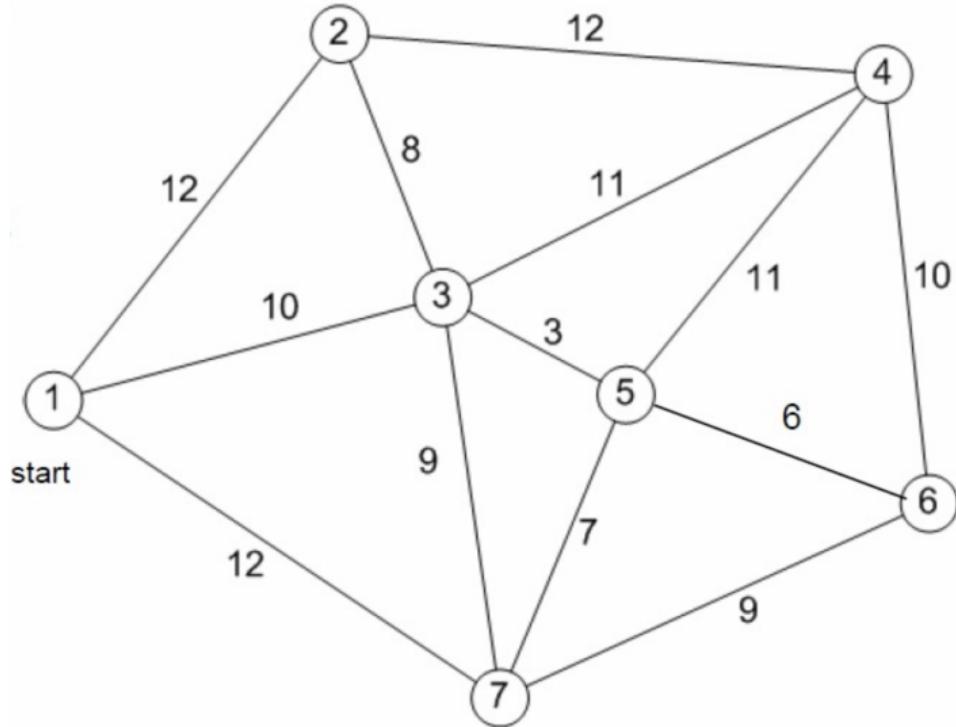
Lesson 3

Graphs

TSP Example 1



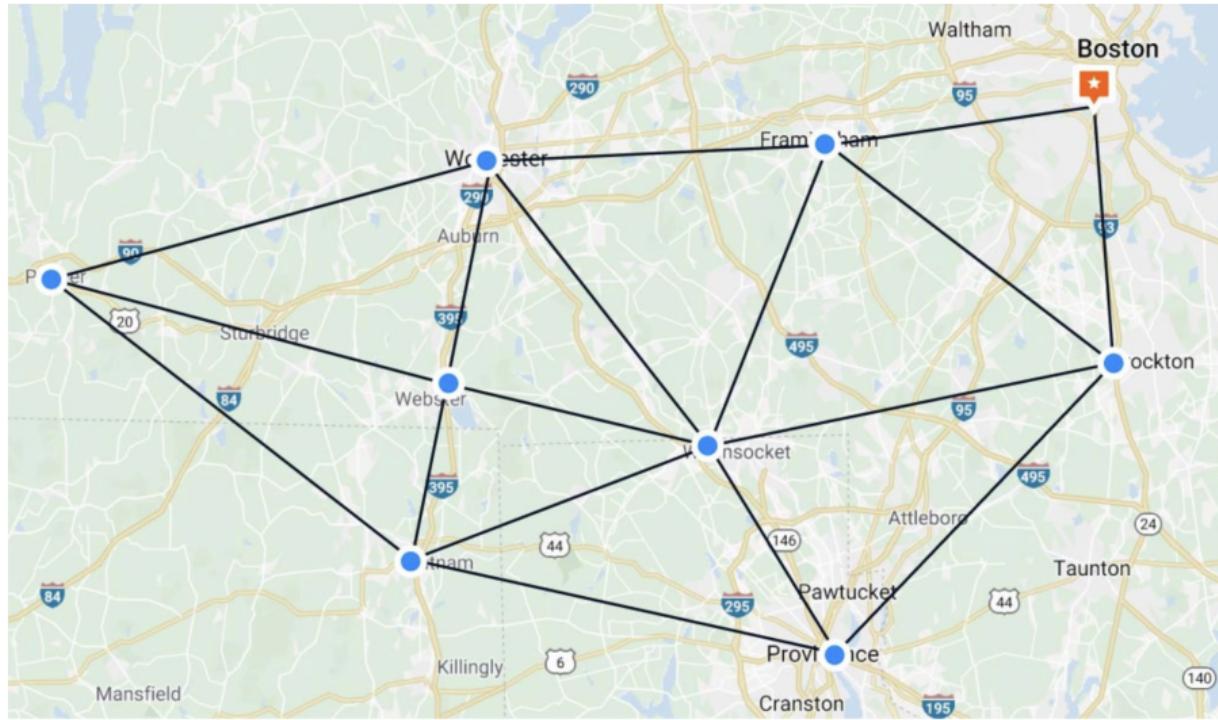
TSP Example 2



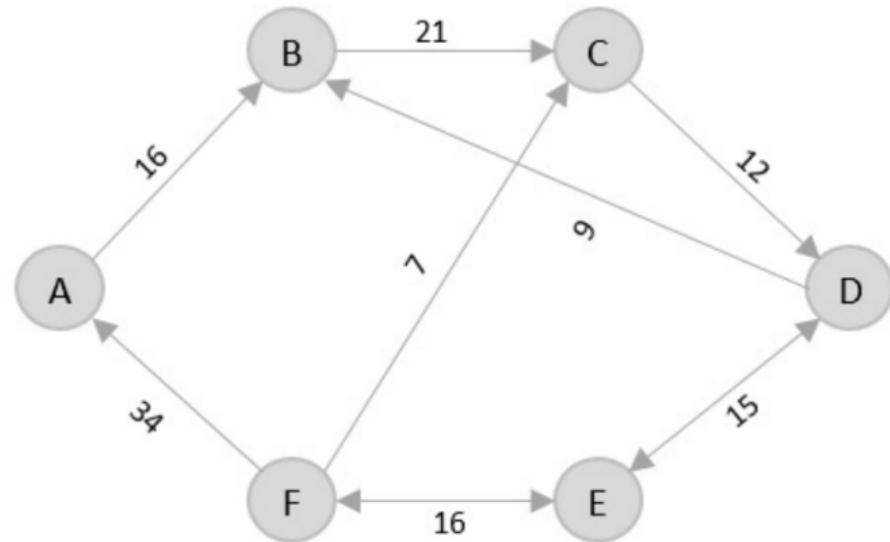
Lesson 3

Graphs

TSP Example 3



TSP Original Problem



Why TSP matters?

It represents real-world problems:

- ▶ Delivery services need to plan routes to drop off packages at multiple locations
- ▶ Logistics companies need to find the shortest path to transport goods efficiently
- ▶ Manufacturing companies need to reduce the travel of robotic arms in factories
- ▶ Solving TSP helps reduce time, fuel costs, and energy, making operations faster and cheaper

Lesson 3

Graphs

How to solve it?

How to solve TSP?

- ▶ Brute-Force Approach
- ▶ Dynamic Programming
- ▶ Approximation Algorithms

Brute-Force Approach

This method explores all possible routes (permutations) between cities, calculates the total distance for each route, and selects the shortest one. It guarantees an optimal solution but is inefficient.

- ▶ Advantages: Guarantees the optimal solution
- ▶ Disadvantages: Extremely slow for large datasets due to factorial growth in possible routes

Dynamic Programming

The travelling salesman problem using dynamic programming breaks the problem into smaller subproblems. It solves these subproblems and stores the results to avoid redundant calculations.

- ▶ Advantages: More efficient than brute-force and provides an exact solution
- ▶ Disadvantages: Still exponential, making it impractical for large numbers of cities, though it performs better than brute-force

Lesson 3

Graphs

How to solve TSP?

Approach	Time Complexity	Optimality	Best for
Brute-Force	$O(n!)$	Guaranteed optimal solution	Small datasets ($n < 10$).
Dynamic Programming (Held-Karp)	$O(n^2 * 2^n)$	Guaranteed optimal solution	Moderate datasets ($n < 20$).
Nearest Neighbor	$O(n^2)$	No guarantee	Quick and simple solutions for small datasets.
Christofides Algorithm	$O(n^3)$	$\leq 1.5 \times$ optimal (metric TSP)	Metric TSP with triangle inequality.

Travelling salesman problem (TSP)

It is an **NP-hard** problem in combinatorial optimization, important in theoretical computer science and operations research.

Applications: the travelling purchaser problem, the vehicle routing problem and the ring star problem are three generalizations of TSP.

NP-Hard problem?

Algorithm Complexity

What is a NP-Hard problem?

Computational problems

There are problems which can be solved simple way and problems which might not be simple to solve at all, but still for which we can find an algorithm to solve them.

And there are problems for which we cant find an algorithm at all to solve them.

Problem classification

Computational complexity theory focuses on classifying computational problems, describing all possible algorithms that could be used to solve them. It tries to classify problems that can or cannot be solved.

Complexity Classes

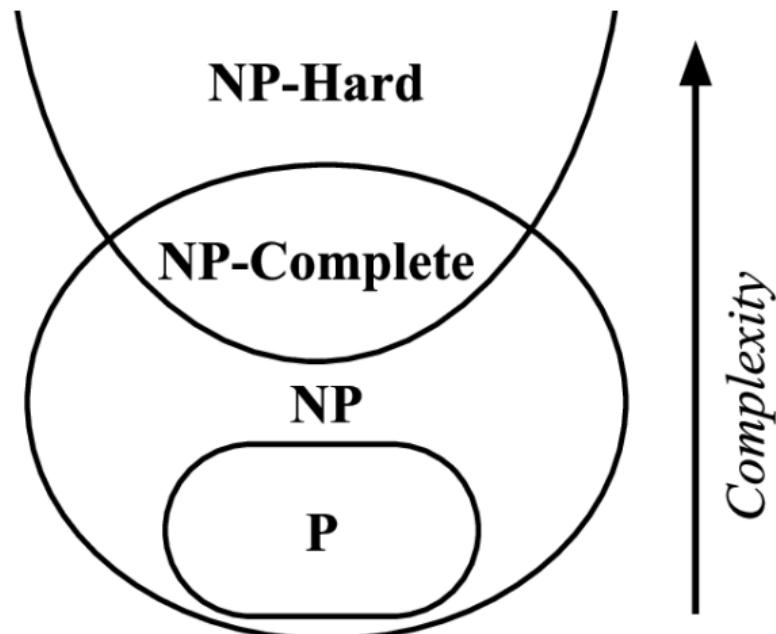
In CS, problems are divided into classes known as Complexity classes. A Complexity Class is a set of problems, which can be solved by computers depending on their related complexity.

- ▶ Problems that can be efficiently solved (Polynomial time)
- ▶ Problems with no efficient solution (Exponential time)
- ▶ Problems that cannot be solved by computers

Lesson 3

Algorithm Complexity

Complexity Classes. Types



Complexity Classes. Types

- ▶ **P class:** Polynomial Time. Problems that can be solved by an algorithm in polynomial time. In other words, the time taken to solve these problems grows at a polynomial rate as the input size increases.
- ▶ **NP class:** Non-deterministic Polynomial Time. Problems in class NP are those where a solution can be verified in polynomial time, but we don't necessarily know how to solve them efficiently.
- ▶ **NP-Hard:** NP-Hard problems are at least as hard as the hardest problems in NP, but they may not even belong to class NP.

Complexity Classes. P class

Problems in class P are those that can be solved by an algorithm in polynomial time. In other words, the time taken to solve these problems grows at a polynomial rate as the input size increases. Polynomial time algorithms are considered efficient and practical for real-world use. **If a problem is in class P, it means we can solve it relatively quickly even for large inputs.**

Complexity Classes. NP class

Problems in class NP are those where a solution can be verified in polynomial time, but we don't necessarily know how to solve them efficiently. The key idea here is that if someone gives us a solution, we can check if it's correct in a reasonable amount of time. **NP problems are crucial because they are often more challenging to solve but still easy to verify. Easy means Polynomial time.**

Lesson 3

Algorithm Complexity

A problem is in NP if it is easy to verify its solutions.

Complexity Classes. NP-Hard class

NP-Hard problems are at least as hard as the hardest problems in NP, but they may not even belong to class NP. **We could develop an algorithm to solve the problem, but once it's executed, it keeps running endlessly without ever stopping. That would only happen if you use a large input for a given problem.**

Lesson 3

Algorithm Complexity

A problem is NP-hard if it is difficult to solve, or find a solution.

Complexity Classes. NP-Hard class

Example: trying to solve a traveling salesman problem and you input 1000 cities, expecting to get a solution, it won't happen. You've essentially set the program to run indefinitely, and it will take about 240 years to solve if you have a super computer.

Complexity Classes

An algorithm having time complexity of the form $O(n^k)$ for input n and constant k is called polynomial time solution. These solutions scale well.

On the other hand, time complexity of the form $O(k^n)$ is exponential time.

Polynomial Time

An algorithm is said to be of polynomial time if its running time is upper bounded by a **polynomial** expression in the size of the input for the algorithm, that is $T(n) = O(n^k)$ for some positive constant k .

What is Polynomial Complexity?

An algorithm is said to have polynomial complexity if its resource usage can be expressed as a polynomial function of the input size, n . Formally, an algorithm has polynomial time complexity if its running time is $O(n^k)$ for some non negative integer k .

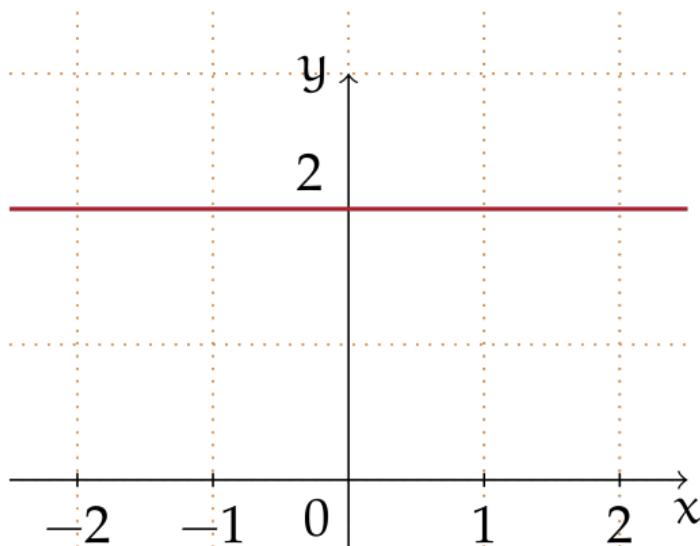
Polynomial expression

In mathematics, a polynomial is a mathematical expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication and exponentiation to nonnegative integer powers, and has a finite number of terms. An example of a polynomial of a single indeterminate x is $x^2 + 3x - 5$

Lesson 3

Algorithm Complexity

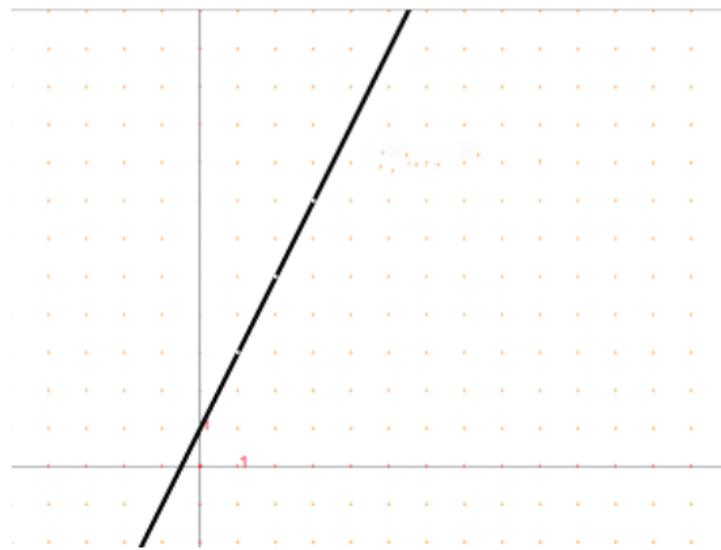
Polynomial of degree 0 $f(x) = 2$



Lesson 3

Algorithm Complexity

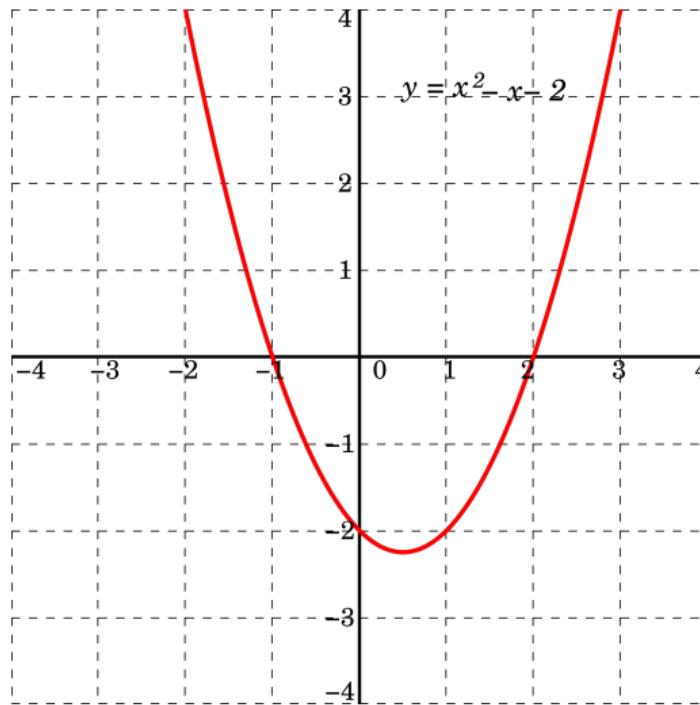
Polynomial of degree 1 $f(x) = 2x + 1$



Lesson 3

Algorithm Complexity

Polynomial of degree 2



Characteristics of Polynomial Complexity

- ▶ Predictable Growth: The growth rate of polynomial functions is predictable and manageable for small to moderate input sizes.
- ▶ Efficient for Larger Inputs: Algorithms with polynomial complexity are generally considered efficient and scalable.
- ▶ Classification: Polynomial time algorithms are categorized based on their degree:
 - Linear Time $O(n)$: The simplest form, where the running time increases directly with the input size.
 - Quadratic Time $O(n^2)$: The running time increases with the square of the input size.

Polynomial Time

Some examples of polynomial-time algorithms:

- ▶ Linear search, selection sort sorting algorithm
- ▶ Bubble sort
- ▶ All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.
- ▶ Maximum matchings in graphs can be found in polynomial time. In some contexts, especially in optimization, one differentiates between strongly polynomial time and weakly polynomial time algorithms.

Exponential Time

An algorithm is said to be exponential time, if $T(n)$ is upper bounded by $2^{poly(n)}$, where poly is some polynomial in n.

What is Exponential Complexity?

An algorithm has exponential complexity if its resource usage can be expressed as an exponential function of the input size, typically $O(2^n)$ or $O(c^n)$ for some constant $c > 1$.

Characteristics of Exponential Complexity

- ▶ Rapid Growth: Exponential functions grow much faster than polynomial functions. Even for relatively small input sizes, the resource requirements can become impractically large.
- ▶ Hard to control: Algorithms with exponential complexity are generally considered hard to control for large input sizes.
- ▶ Resource Exhaustion: Due to the rapid growth in resource requirements, these algorithms often become infeasible as the input size increase.

Algorithm Complexity

A computational problem is a task solved by a computer.
A computation problem is solvable by an algorithm.

Computational complexity theory focuses on classifying computational problems according to their resource usage, and explores the relationships between these classifications

Algorithm Complexity

A problem is regarded as difficult if its solution requires significant resources, whatever the algorithm used. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying their computational complexity, i.e., the amount of resources needed to solve them, such as time and storage.

What is Algorithm Complexity Analysis?

The primary motive to use data structures and algorithms (DSA) is to **solve** a problem effectively and efficiently. But how can you decide if a program is efficient or not? We measure that by analysing the algorithm complexity.

Algorithm Complexity

The **time complexity** of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer (the running time of your algorithm)

The **space complexity** of an algorithm describes how much memory is required for the algorithm to operate.

Space complexity

The space complexity of an algorithm is the space taken by an algorithm to run the program for a given input size. The program has some space requirements necessary for its execution these include auxiliary space and input space.

the space taken by the algorithm to run for a given input size

Time complexity

Time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Time complexity

Time algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n . If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n .

Time complexity

To be able to select the best algorithm, calculation of complexity and time consumption of an algorithm is important. That's why we need to take into consideration time complexity analysis.

Time complexity

There are 3 type of time complexity analysis (notations):

- ▶ **Big-O:** Denotes the upper bound of any algorithm's runtime, the time is taken by the algorithm in the worst case.
- ▶ **Big-Omega:** Denotes the best runtime of an algorithm
- ▶ **Big-Theta:** Denotes average case time complexity

Analysis of algorithms

Analyzing the amount of resources needed by a particular algorithm to solve a problem.

Complexity Classes

The common resources required by a solution are **time** and **space**, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

Big-O notation

Big-O Notation

Big-O notation is the most common notation to represent algorithmic complexity. It gives an upper bound on complexity and hence it signifies the worst-case performance of the algorithm. With such a notation, it's easy to compare different algorithms because the notation tells clearly how the algorithm scales when input size increases. This is often called the order of growth.

Big-O notation helps us answer the question,
“Will it scale?”

Describes the upper bound of any algorithm's runtime, example the time is taken by the algorithm in the worst case.

Constant time $O(1)$ - Excellent/Best

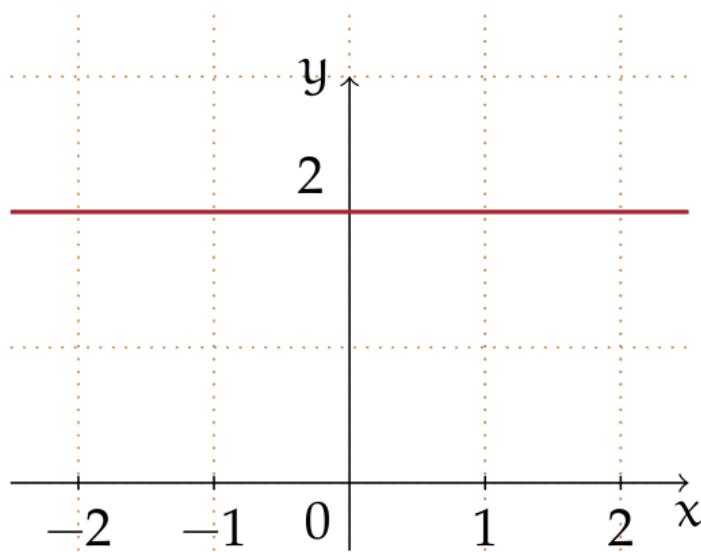
The algorithm runs for the same amount of time every time irrespective of the input size. We call this, constant time complexity.

For example, accessing an element in an array by index, searching for the first word in a dictionary. Same way, joining the end of a queue in a bank is of constant complexity regardless of how long the queue is.

Lesson 3

Algorithm Complexity

Constant time $O(1)$



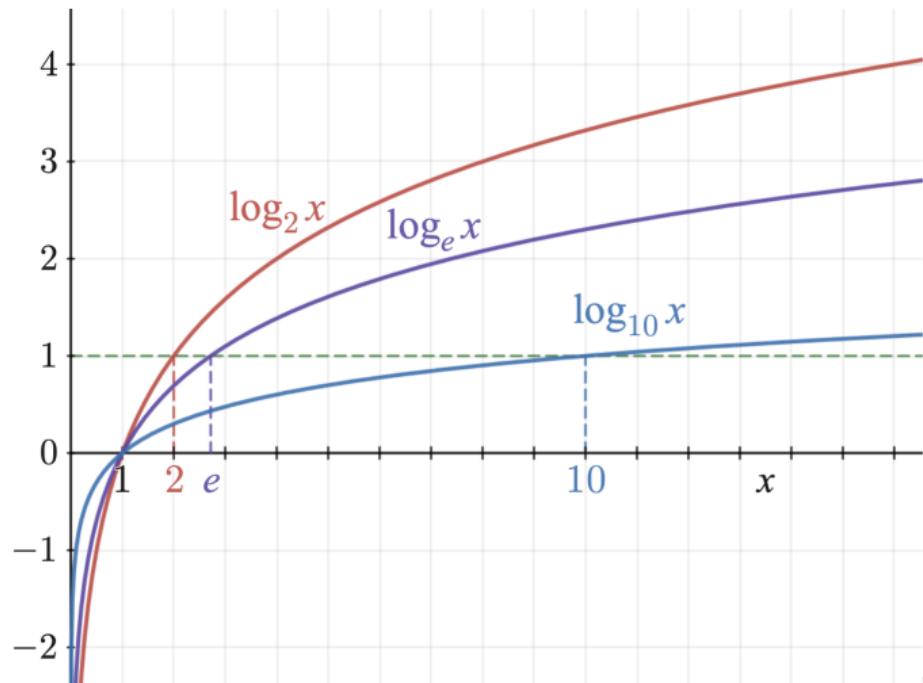
Logarithmic time $O(\log n)$ - Good

When the algorithm runtime increases very slowly compared to an increase in input size, we say the algorithm is of logarithmic time complexity.

Lesson 3

Algorithm Complexity

Logarithmic time $O(\log n)$ - Good



Linear time - $O(n)$ - Fair

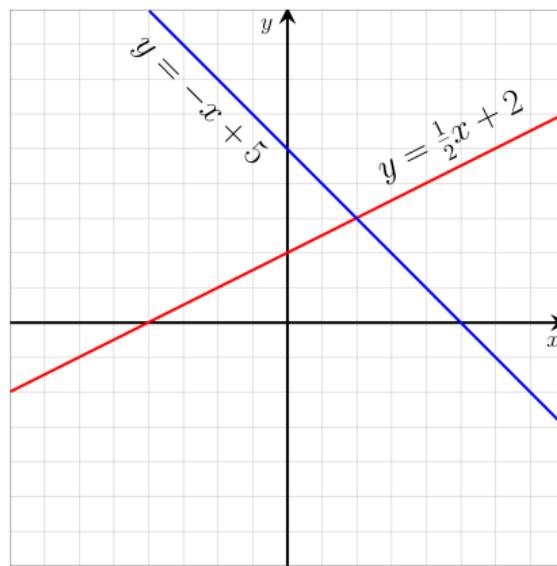
If the algorithm runtime is linearly proportional to the input size then we call the algorithm to be of linear time complexity.

We are looking for a specific item in a long unsorted list. We will probably compare each item. Search time is proportional to the list size. Here we say the complexity is to be **linear**.

Lesson 3

Algorithm Complexity

Linear time - $O(n)$ - Fair



Lesson 3

Algorithm Complexity

Log-Liner time $O(n \log n)$ - Bad

Quadratic time $O(n^2)$ - Very bad

If we are planning to find all duplicates in an unsorted list then the complexity becomes quadratic. Checking for duplicates for one item is of linear complexity. But if we do this for all items, complexity becomes quadratic. Similarly, if all people in a room are asked to shake hands with every other person, the complexity **quadratic**.

Lesson 3

Algorithm Complexity

Exponential time $O(2^n)$ - Horrible

Lesson 3

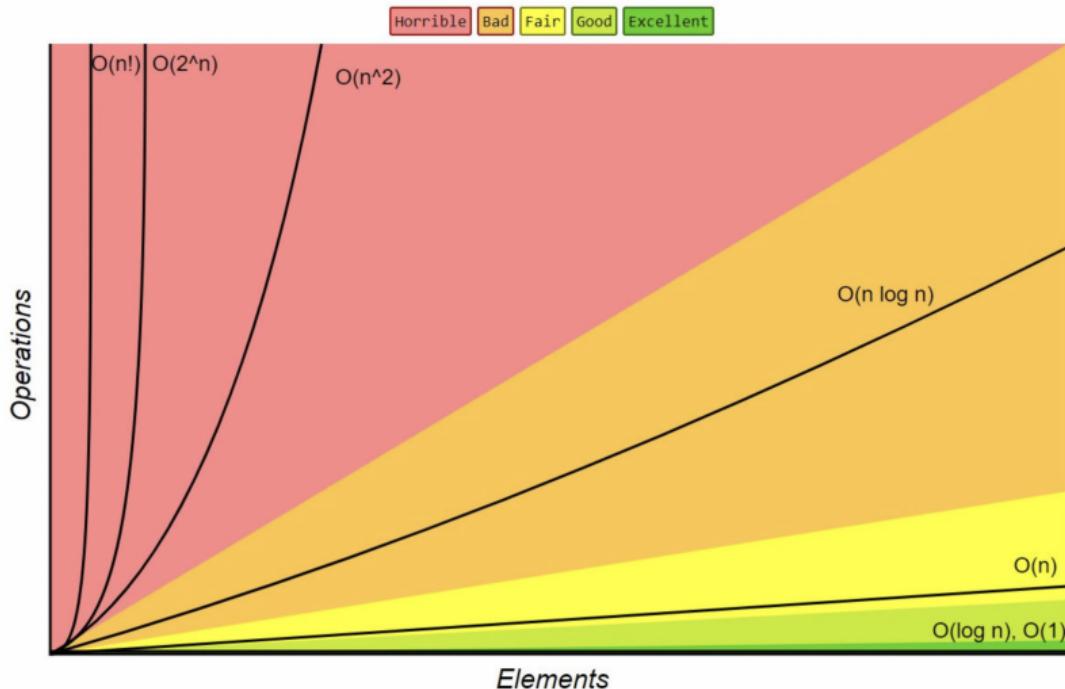
Algorithm Complexity

Factorial time $O(n!)$ - Worst

Lesson 3

Algorithm Complexity

Time complexity



Travelling salesman problem (TSP). NP-Hard

TSP is considered an NP-hard problem, meaning that the time required to find the optimal solution increases exponentially as the number of cities grows.

Travelling salesman problem (TSP). Brute-force

If we start to solve this problem using the brute-force method is SLOW. Very slow.

Brute force? Involves evaluating every possible permutation of cities to identify the shortest route. This method has a factorial runtime $O(n!)$, making it impractical even for moderately sized datasets. For example, with just 20 cities, there are over 2.4 quintillion (10^{18}) possible routes!

Travelling salesman problem (TSP). Brute-force

Suitable only for a small network with few cities

Travelling salesman problem (TSP). Dynamic Programming

The Held-Karp Algorithm

It offers a more efficient solution using dynamic programming. It reduces the time complexity to $O(n^2 * 2^n)$, which is significantly better than brute force but still exponential. While feasible for smaller datasets, the Held-Karp algorithm becomes impractical as the number of cities grows.

Travelling salesman problem (TSP). Dynamic programming

Suitable for an average network with aprox 50 cities

Travelling salesman problem (TSP). Polynomial Time

The holy grail of TSP research is a polynomial-time algorithm—one that can solve the problem efficiently as the number of cities increases. Despite extensive efforts, no such algorithm exists for the general case of the TSP. This remains research and development, in the context of the P vs NP problem.

Why should we care about an algorithm's performance when we have cloud and processors are getting faster and memories are getting cheaper and cheaper?

Solving the problem!

Complexity is about the algorithm itself, the way it processes the data to solve a given problem. It's a software design concern at the "idea level". It is NOT about Python vs Java execution time.

Lesson 3

Algorithm Complexity

How efficient you solve the problem?

Lesson 3

Algorithm Complexity

It's about how fast and efficient you solve a problem.

It's possible to have an inefficient algorithm that's executed on high-end hardware to give a result quickly. However, with large input datasets, the limitations of the hardware will become apparent. Thus, it's desirable to optimize the algorithm first before thinking about hardware upgrades.

Sorting Algorithms

Lesson 3

Sorting Algorithms

8 1 10 9 7

How could we sort this list of numbers?

Lesson 3

Sorting Algorithms

8 1 10 9 7

Sort them numerically: 1 7 8 9 10

Lesson 3

Sorting Algorithms

8 1 10 9 7

Or sort them alpha-numerically: 1 10 7 8 9

Lesson 3

Sorting Algorithms

8 1 10 9 7

Do we need extra memory to sort or not? In place sort does not require extra memory. Out of place sort does require extra memory.

Lesson 3

Sorting Algorithms

8 1 10 1 7 8 9

How are we gonna sort these numbers, which have duplicates?

Lesson 3

Sorting Algorithms

$8^1 1^1 10 1^2 7 8^2 9$

We can sort these numbers by $1^1 1^2 7 8^1 8^2 9 10$ keeping the order of the duplicates. We call this a stable sort.

Lesson 3

Sorting Algorithms

We need to think how to sort them

Lesson 3

Sorting Algorithms

Sorts

- ▶ numerically or not
- ▶ in place vs out of place
- ▶ stable or unstable sorting
- ▶ the complexity

Lesson 3

Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$
Count Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
Shell Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(1)$

Selection Sort

- ▶ in-place comparison-based algorithm
- ▶ repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element
- ▶ repeats until the entire array is sorted
- ▶ inefficient on large lists

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n)$ swaps
Best-case performance	$O(n^2)$ comparisons, $O(1)$ swap
Average performance	$O(n^2)$ comparisons, $O(n)$ swaps
Worst-case space complexity	$O(1)$ auxiliary
Optimal	No

Selection Sort. Time complexity $O(n^2)$

There are two nested loops!

One loop to select n element of array one by one = $O(n)$

Another loop to compare that element with every other array element = $O(n)$

$$O(n) * O(n) = O(n * n) = O(n^2)$$

Advantages of Selection Sort

- ▶ **easy:** very good for teaching basic sorting concepts. runtime, the time is taken by the algorithm in the worst case.
- ▶ **memory efficient:** it requires constant $O(1)$ extra memory space
- ▶ **simple:** good when we dont have memory constraints
- ▶ Suitable for small lists

Disadvantages of Selection Sort

- ▶ **slow**: it has $O(n^2)$ time complexity vs others
- ▶ **not stable**: does not maintain the relative order of equal elements which means it is not **stable**.

Lesson 3

Sorting Algorithms

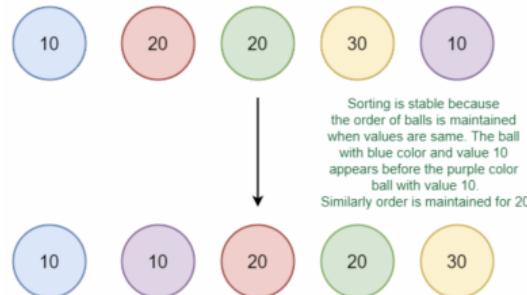
But what is a stable sorting algorithm?

Lesson 3

Sorting Algorithms

Stable sorting algorithm

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set



```
def selection_sort (a):
    n = len(a)
    for i in range(n - 1):

        # saves the minimum element
        min_index = i

        # Iterate through the unsorted array to find the actual minimum
        for j in range(i + 1, n):
            if a[j] < a[min_index]:

                # Update min_idx if a smaller element is found
                min_index = j

        # Swap minimum element to its correct position
        a[i], a[min_index] = a[min_index], a[i]
```

Insertion Sort

- ▶ Start with 2nd element of the array as 1st element is assumed to be sorted
- ▶ Compare 2nd element with the 1st element and check if the 2nd element is smaller then swap them.
- ▶ Move to the third element and compare it with the first two elements and put at its correct position
- ▶ Repeat until the entire array is sorted.

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary
Optimal	No

Lesson 3

Sorting Algorithms

Insertion Sort. Time complexity $O(n^2)$

Best case $O(n)$ if the list is already sorted, where n is the number of elements in the list. Worst case $O(n^2)$

Advantages of Insertion Sort

- ▶ **simple:** very simple and easy to implement runtime, the time is taken by the algorithm in the worst case.
- ▶ **memory efficient:** it requires constant $O(1)$ extra memory space
- ▶ **stable:** it is a stable sorting algorithm
- ▶ Suitable for small or almost sorted lists

Disadvantages of Insertion Sort

- ▶ **inefficient**: slow for large lists
- ▶ **slow**: not as efficient as other sorting algorithms: merge or quicksort

```
def insertionSort (a):
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1

        # Swap elements of a[0..i-1], greater than key 1 position ahead
        while j >= 0 and key < a[j]:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
```

Bubble Sort

- ▶ very simple algorithm
- ▶ repeatedly steps through the input list element by element
- ▶ comparing the current element with the one after it
- ▶ swapping their values if needed
- ▶ these passes through the list are repeated until no swaps have to be performed during a pass, meaning that the list has become fully sorted

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary
Optimal	No

Bubble Sort. Time complexity $O(n^2)$

Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted.

Advantages of Bubble Sort

- ▶ **easy:** easy to understand and implement runtime, the time is taken by the algorithm in the worst case.
- ▶ **memory efficient:** it requires constant $O(1)$ extra memory space
- ▶ **stable:** it is a stable sorting algorithm
- ▶ Suitable for small or almost sorted lists

Disadvantages of Bubble Sort

- ▶ **inefficient**: slow for large data sets
- ▶ **slow**: not as efficient as other sorting algorithms: merge or quicksort.

```
def bubbleSort( arr ):
    n = len(a)

    # Traverse the array a
    for i in range(n):
        swapped = False

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # Traverse the array a from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
                swapped = True

            if (swapped == False):
                break
```

Lesson 3

Sorting Algorithms

Heap Sort

- ▶ an efficient comparison-based sorting algorithm
- ▶ reorganizes an input array into a tree (heap)
- ▶ a data structure where each node is greater than its children
- ▶ repeatedly removes the largest node from that heap, placing it at the end of the array

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$O(n \log n)$ (distinct keys) ^{[1][2]} or $O(n)$ (equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ total $O(1)$ auxiliary

Heap Sort. Time complexity $O(n \log n)$

Better than bubblesort, the heapsort has a very simple implementation and a more favorable worst-case $O(n \log n)$ runtime. Memory Space: $O(\log n)$, due to the recursive call stack. However, the space can be $O(1)$ for iterative implementation.

Advantages of Heap Sort

- ▶ **efficient:** good for sorting large data sets runtime, the time is taken by the algorithm in the worst case.
- ▶ **memory usage:** Memory usage can be minimal (by writing an iterative heapify() instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work

Disadvantages of Heap Sort

- ▶ **Unstable:** it is an unstable algorithm

```
def heapify( arr , n , i):  
  
    # Initialize largest as root  
    largest = i  
  
    # left index = 2*i + 1  
    l = 2 * i + 1  
  
    # right index = 2*i + 2  
    r = 2 * i + 2  
  
    # If left child is larger than root  
    if l < n and arr[l] > arr[ largest ]:  
        largest = l
```

```
# If right child is larger than largest so far
if r < n and arr[r] > arr[ largest ]:
    largest = r

# If largest is not root
if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i] # Swap

# Recursively heapify the affected sub-tree
heapify( arr , n, largest )
```

```
def heapSort(arr ):  
  
n = len(arr)  
  
# Build heap (rearrange array)  
for i in range(n // 2 - 1, -1, -1):  
    heapify(arr , n, i)  
  
# One by one extract an element from heap  
for i in range(n - 1, 0, -1):  
  
    # Move root to end  
    arr [0], arr [i] = arr[i], arr [0]  
  
# Call max heapify on the reduced heap  
heapify(arr , i , 0)
```

Quicksort

- ▶ an efficient, general-purpose algorithm
- ▶ faster than merge sort, heapsort
- ▶ a **divide-and-conquer** algorithm
- ▶ select a **pivot** and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot

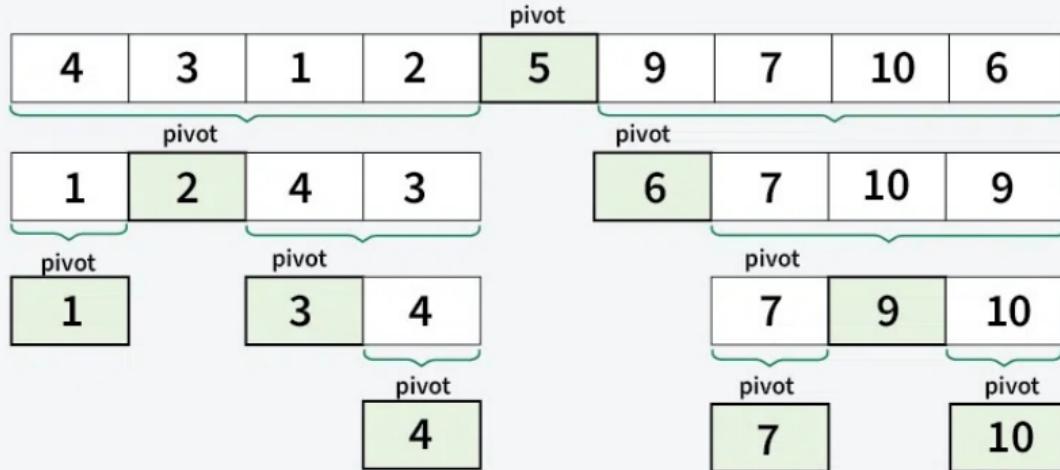
Class	Sorting algorithm
Worst-case performance	$O(n^2)$
Best-case performance	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ auxiliary (naive) $O(\log n)$ auxiliary (Hoare 1962)

Lesson 3

Sorting Algorithms

Quicksort

Here, we have represented the recursive call after each partitioning step of the array.



Lesson 3

Sorting Algorithms

But what is divide-and-conquer?

Divide and Conquer

- ▶ its an algorithm **design paradigm**
- ▶ a generic **model** or framework which underlies the design of a class of algorithms
- ▶ there are different models or frameworks in how we design algorithms

Lesson 3

- ▶ Backtracking
- ▶ Branch and bound
- ▶ Brute-force search
- ▶ **Divide and conquer**
- ▶ Dynamic programming
- ▶ Greedy algorithm
- ▶ Recursion
- ▶ Prune and search

Divide and Conquer

- ▶ Breaks down a problem into two or more sub-problems
- ▶ sub-problems of the same or related type
- ▶ Repeat until these become simple enough to be solved directly
- ▶ Then the solutions to the sub-problems are then combined to give a solution to the original problem

Divide and Conquer

The **divide-and-conquer** technique is the basis of efficient algorithms for many problems, such as sorting (quicksort, merge sort) or multiplying large numbers, finding the closest pair of points, syntactic analysis, and computing the discrete Fourier transform(FFT).

Divide and Conquer

Designing efficient divide-and-conquer algorithms can be difficult

It is often necessary to generalize the problem to make it solvable to a **recursive** solution. The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Lesson 3

Sorting Algorithms

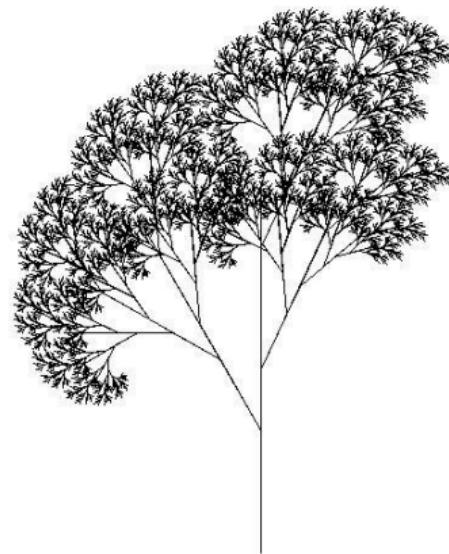
Do you remember **Recursion?**

Recursion

In computer science (CS), **recursion** is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem.

Recursion

Recursion solves such recursive problems by using functions that call themselves from within their own code. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.



- ▶ **Backtracking**
- ▶ Branch and bound
- ▶ Brute-force search
- ▶ Divide and conquer
- ▶ Dynamic programming
- ▶ Greedy algorithm
- ▶ Recursion
- ▶ Prune and search

Backtracking

- ▶ another algorithm **design paradigm**
- ▶ for different type of algorithms
- ▶ for which the solution is found by checking and dropping candidates for the solution

Backtracking

Backtracking is a technique for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Backtracking

Backtracking can be applied only for problems which admit the concept of a **partial candidate solution** and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute-force enumeration of all complete candidates, since it can eliminate many candidates with a single test.

Backtracking

Backtracking is an important tool for solving **constraint satisfaction problems**, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing, for the knapsack problem and other combinatorial optimization problems.

Backtracking. Constraint satisfaction problems

Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods.

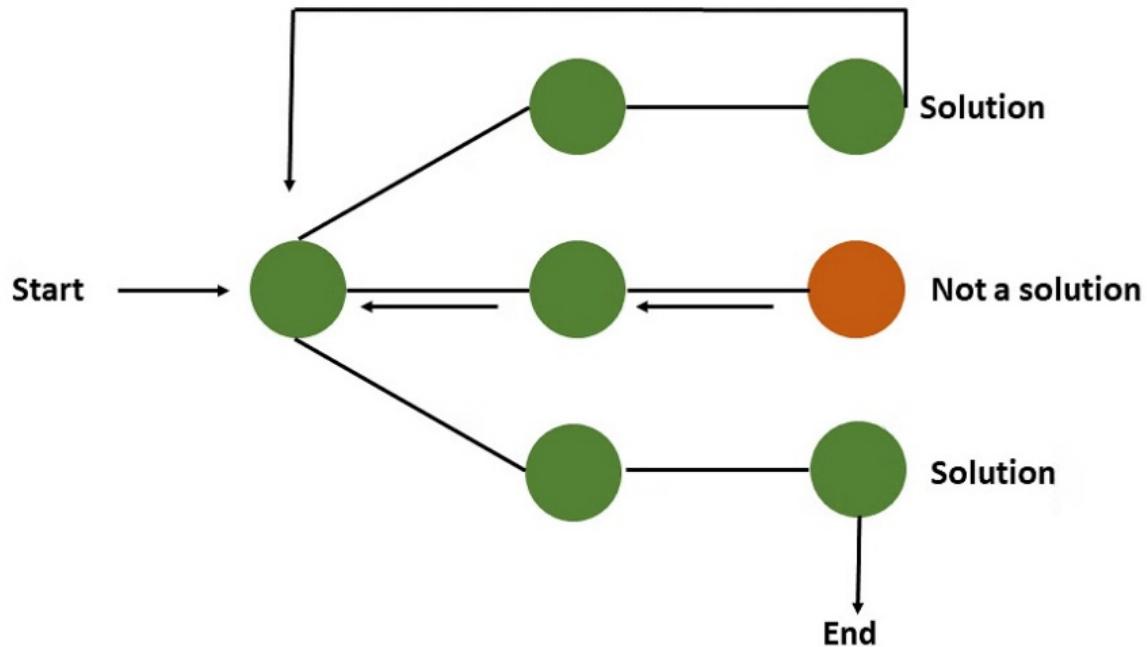
Backtracking

The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.

Lesson 3

Sorting Algorithms

Backtracking



Backtracking

Conceptually, the partial candidates are represented as the nodes of a tree structure, the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.

Quicksort. Time complexity $O(n \log n)$

Slightly faster than merge sort and heapsort for randomized data, particularly on larger distributions.

Worst-case $O(n^2)$

Advantages of Quicksort

- ▶ **Efficient:** It is a divide-and-conquer algorithm that makes it easier to solve problems. It is efficient on large data sets.
- ▶ **Low overhead:** it requires a small amount of memory to function

Disadvantages of Quicksort

- ▶ **Worst complexity:** $O(n^2)$ when the pivot is chosen poorly
- ▶ **Not efficient** small datasets
- ▶ **Not stable** meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions)

```
# Partition function
def partition ( arr , low , high ):

    # Choose the pivot
    pivot = arr[high]

    # Index of smaller element and indicates
    # the right position of pivot found so far
    i = low - 1
```

```
# Traverse arr[low.. high] and move all smaller
# elements to the left side. Elements from low to
# i are smaller after every iteration
for j in range(low, high):
    if arr[j] < pivot:
        i += 1
        swap(arr, i, j)

# Move pivot after smaller elements and
# return its position
swap(arr, i + 1, high)
return i + 1
```

Swap function

```
def swap(arr, i, j):  
    arr[i], arr[j] = arr[j], arr[i]
```

```
# The QuickSort function implementation
def quickSort( arr , low , high ):
    if low < high:

        # pi is the partition return index of pivot
        pi = partition( arr , low , high )

        # Recursion calls for smaller elements
        # and greater or equals elements
        quickSort( arr , low , pi - 1 )
        quickSort( arr , pi + 1 , high )
```

```
arr = [10, 7, 8, 9, 1, 5]
```

```
n = len(arr)
```

```
quickSort( arr , 0, n - 1)
```

```
for val in arr:  
    print(val, end=",")
```

Merge Sort

- ▶ an efficient comparison-based sorting algorithm
- ▶ a stable sort
- ▶ a divide-and-conquer algorithm
- ▶ invented by John von Neumann 1945

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$\Omega(n \log n)$ typical, $\Omega(n)$ natural variant
Average performance	$\Theta(n \log n)$
Worst-case space complexity	$O(n)$ total with $O(n)$ auxiliary, $O(1)$ auxiliary with linked lists ^[1]

Merge Sort

- ▶ Divide the unsorted list into n sub-lists, each containing one element (a list of one element is considered sorted).
- ▶ Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list

Merge Sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

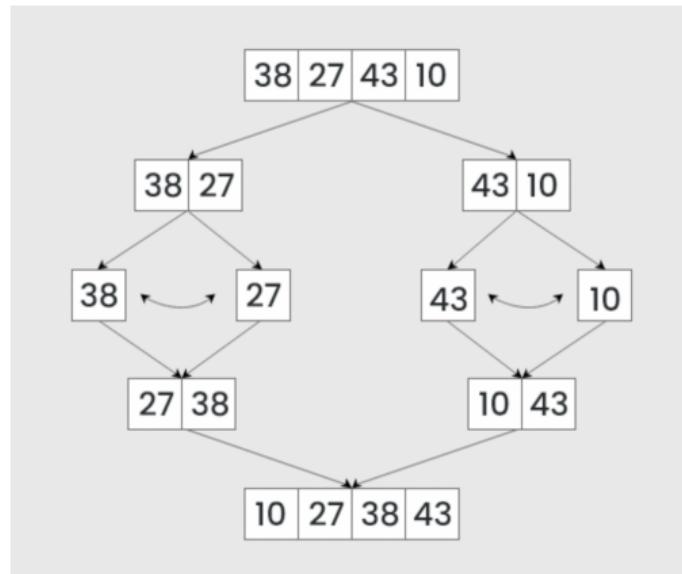
Merge Sort

The merge sort algorithm is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Lesson 3

Sorting Algorithms

Merge sort



Merge Sort. Time complexity $O(n \log n)$

Best Case: $O(n \log n)$, when the array is already sorted or nearly sorted. Worst Case: $O(n \log n)$, when the array is sorted in reverse order. Memory space: $O(n)$, additional space is required for the temporary array used during merging.

Advantages of Merge Sort

- ▶ **Stability:** it is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- ▶ **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of $O(N\log N)$, which means it performs well even on large datasets.
- ▶ **Simplicity:** Using a divide-and-conquer approach

Disadvantages of Merge Sort

- ▶ **Space complexity:** extra memory for sorting
- ▶ **Not in-place:** it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- ▶ **Slow:** slower than quicksort

```
def merge(arr, left , mid, right ):
    n1 = mid - left + 1
    n2 = right - mid

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[ left + i]
    for j in range(n2):
        R[j] = arr[ mid + 1 + j]
```

```
i = 0 # Initial index of first subarray
j = 0 # Initial index of second subarray
k = left # Initial index of merged subarray

# Merge the temp arrays back
# into arr[left .. right]
while i < n1 and j < n2:
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1
```

Copy the remaining elements of L[],

if there are any

while i < n1:

 arr [k] = L[i]

 i += 1

 k += 1

Copy the remaining elements of R[],

if there are any

while j < n2:

 arr [k] = R[j]

 j += 1

 k += 1

```
def merge_sort( arr , left , right ):
    if left < right:
        mid = (left + right) // 2

        merge_sort( arr , left , mid)
        merge_sort( arr , mid + 1, right )
        merge(arr, left , mid, right )
```

```
def print_list ( arr ):
    for i in arr:
        print(i, end="-")
    print()
```

```
arr = [12, 11, 13, 5, 6, 7]
print("Given-array-is")
print_list (arr)

merge_sort( arr , 0, len( arr ) - 1)

print("\\nSorted-array-is")
print_list (arr)
```

Searching Algorithms

Searching Algorithms

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. In this tutorial, we are mainly going to focus upon searching in an array. When we search an item in an array, there are two most common algorithms used based on the type of input array.

Searching algorithms

- ▶ **linear search:** used for an unsorted array. It mainly does one by one comparison of the item to be search with array elements. It takes linear or $O(n)$ time.
- ▶ **binary search:** It is used for a sorted array. It mainly compares the array's middle element first and if the middle element is same as input, then it returns. Otherwise it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater). This algorithm is faster than linear search and takes $O(\log n)$ time.

Linear search

- ▶ is a method for finding an element within a list
- ▶ it sequentially checks each element of the list until a match is found or the whole list has been searched

Class	Search algorithm
Worst-case performance	$O(n)$
Best-case performance	$O(1)$
Average performance	$O(n)$
Worst-case space complexity	$O(1)$ iterative
Optimal	Yes

Linear search

In **Linear Search**, we iterate over all the elements of the array and check if it the current element is equal to the target element. If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found.

Linear search is also known as sequential search.

```
def search( arr , N, x):  
  
    for i in range(0, N):  
        if ( arr [i] == x):  
            return i  
return -1
```

```
arr = [2, 3, 4, 10, 40]  
x = 10  
N = len(arr)
```

```
# Function call  
result = search(arr , N, x)  
if ( result == -1):  
    print(" Element-is-not-present-in-array")  
else:  
    print(" Element-is-present-at-index" , result )
```

Lesson 3

Searching Algorithms

Binary search

- ▶ known as half-interval search, logarithmic search, or binary chop
- ▶ a search algorithm that finds the position of a target value within a sorted array

Class	Search algorithm
Data structure	Array
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$
Optimal	Yes

Binary search

Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log N)$.

Binary search

Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

```
# It returns location of x in given array arr
def binarySearch(arr , low , high , x):
```

```
    while low <= high:
```

```
        mid = low + (high - low) // 2
```

```
        # Check if x is present at mid
```

```
        if arr [mid] == x:
            return mid
```

```
        # If x is greater, ignore left half
```

```
        elif arr [mid] < x:
            low = mid + 1
```

```
        # If x is smaller, ignore right half
```

```
        else:
```

```
            high = mid - 1
```

```
# If we reach here, then the element
```

```
" " is not present
```

```
arr = [2, 3, 4, 10, 40]
x = 10
```

```
# Function call
result = binarySearch(arr, 0, len(arr)-1, x)
if result != -1:
    print("Element-is-present-at-index", result)
else:
    print("Element-is-not-present-in-array")
```