# Python

## What is a Numpy array?

[**NumPy**](https://www.geeksforgeeks.org/python-numpy/) is the fundamental package for scientific computing in [Python](https://www.geeksforgeeks.org/python-programming-language/). Numpy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences. Numpy is not another programming language but a Python extension module. It provides fast and efficient operations on arrays of homogeneous data.

**Some important points about Numpy arrays:**

* We can create an N-dimensional array in Python using [Numpy.array().](https://www.geeksforgeeks.org/basics-of-numpy-arrays/)
* The array is by default Homogeneous, which means data inside an array must be of the same Datatype. (Note You can also create a structured array in Python).
* Element-wise operation is possible.
* Numpy array has various functions, methods, and variables, to ease our task of matrix computation.
* Elements of an array are stored contiguously in memory. For example, all rows of a two-dimensioned array must have the same number of columns. A three-dimensional array must have the same number of rows and columns on each card.

## What is Python List?

A **Python** [**list**](https://www.geeksforgeeks.org/python-list/) is a collection that is ordered and changeable. In Python, lists are written with square brackets.

**Some important points about Python Lists:**

* The list can be homogeneous or heterogeneous.
* Element-wise operation is not possible on the list.
* Python list is by default 1-dimensional. But we can create an N-Dimensional list. But then too it will be 1 D list storing another 1D list
* Elements of a list need not be contiguous in memory.

## Comparison between Numpy array and Python List

**Python Lists**

1. **Element Overhead:** Lists in Python store additional information about each element, such as its type and reference count. This overhead can be significant when dealing with a large number of elements.
2. **Datatype:** Lists can hold different data types, but this can decrease memory efficiency and slow numerical operations.
3. **Memory Fragmentation:** Lists may not store elements in contiguous memory locations, causing memory fragmentation and inefficiency.
4. **Performance:** Lists are not optimized for numerical computations and may have slower mathematical operations due to Python’s interpretation overhead. They are generally used as general-purpose data structures.
5. **Functionality:** Lists can store any data type, but lack specialized NumPy functions for numerical operations.

**Numpy Arrays**

1. **Homogeneous Data:** NumPy arrays store elements of the same data type, making them more compact and memory-efficient than lists.
2. **Fixed Data Type:** NumPy arrays have a fixed data type, reducing memory overhead by eliminating the need to store type information for each element.
3. **Contiguous Memory:** NumPy arrays store elements in adjacent memory locations, reducing fragmentation and allowing for efficient access.
4. **Array Metadata:** NumPy arrays have extra metadata like shape, strides, and data type. However, this overhead is usually smaller than the per-element overhead in lists.
5. **Performance:** NumPy arrays are optimized for numerical computations, with efficient element-wise operations and mathematical functions. These operations are implemented in C, resulting in faster performance than equivalent operations on lists.

**Memory consumption between Numpy array and lists**

In Python, a list is a built-in data structure that can hold elements of varying data types. However, the flexibility of lists comes at the cost of memory efficiency.

Python’s NumPy library supports optimized numerical array and matrix operations.

# Complexity

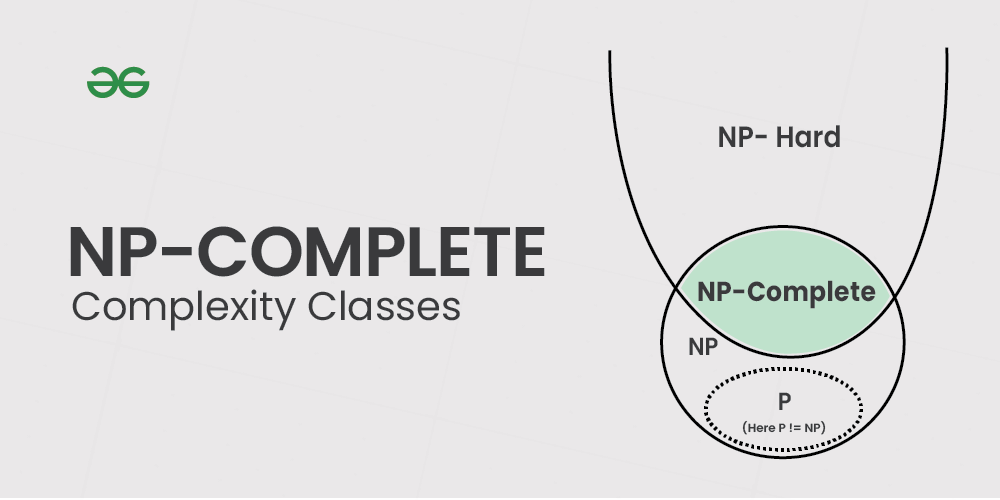
## NP-Complete Complexity Classes

**NP-complete** problems are a subset of the larger class of**NP (nondeterministic polynomial time) problems**. **NP** problems are a class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic Machine. A problem **L** in **NP** is**NP-complete** if all other problems in **NP** can be reduced to**L**in polynomial time. If any**NP-complete** problem can be solved in polynomial time, then every problem in **NP** can be solved in polynomial time. **NP-complete** problems are the hardest problems in the **NP** set.

A decision problem **L** is **NP-complete** if it follow the below two properties:

1. **L** is in **NP** (Any solution to NP-complete problems can be checked quickly, but no efficient solution is known).
2. Every problem in **NP** is reducible to **L** in polynomial time (Reduction is defined below).

A problem is **NP-Hard** if it obeys Property 2 above and need not obey Property 1. Therefore, a problem is **NP-complete** if it is both **NP** and **NP-hard**.



### Famous Examples of NP-Complete Problems

Let’s explore some of the most well-known NP-complete problems:

**1. The Traveling Salesman Problem (TSP)**

The TSP asks: “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 starting city?”

This problem has applications in logistics, planning, and microchip manufacturing. While it’s easy to verify a given solution, finding the optimal route becomes exponentially more difficult as the number of cities increases.

**2. Boolean Satisfiability Problem (SAT)**

SAT is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. For example:

(x1 OR x2) AND (NOT x1 OR x3) AND (NOT x2 OR NOT x3)

The challenge is to find values for x1, x2, and x3 that make the entire formula true, or determine that no such values exist.

**3. Graph Coloring Problem**

This problem asks whether the vertices of a given graph can be colored using at most k colors, such that no two adjacent vertices share the same color. It has applications in scheduling, register allocation in compilers, and even solving Sudoku puzzles.

**4. Subset Sum Problem**

Given a set of integers and a target sum, the subset sum problem asks whether there exists a subset of the integers that adds up exactly to the target sum. This problem is related to various optimization and decision-making scenarios.

# Sorting Algorithms

## Counting sort

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **counting sort** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm) a collection of objects according to keys that are small positive [integers](https://en.wikipedia.org/wiki/Integer); that is, it is an [integer sorting](https://en.wikipedia.org/wiki/Integer_sorting) algorithm. It operates by counting the number of objects that possess distinct key values, and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

**function** CountingSort(input, *k*)

count ← array of *k* + 1 zeros

output ← array of same length as input

**for** *i* = 0 **to** length(input) - 1 **do**

*j* = key(input[*i*])

count[*j*] = count[*j*] + 1

**for** *i* = 1 **to** *k* **do**

count[*i*] = count[*i*] + count[*i* - 1]

**for** *i* = length(input) - 1 **down to** 0 **do**

*j* = key(input[*i*])

count[*j*] = count[*j*] - 1

output[count[*j*]] = input[*i*]

**return** output

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore, the time for the whole algorithm is the sum of the times for these steps, **O(n + k).**

# Trees

## Binary Tree

**Binary Tree**is a **non-linear**and**hierarchical**data structure where each node has at most two children referred to as the **left child** and the**right child**.  The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**.

Each node in a Binary Tree has three parts:

* Data
* Pointer to the left child
* Pointer to the right child

**Terminologies in Binary Tree**

* **Nodes:** The fundamental part of a binary tree, where each node contains **data**and **link**to two child nodes.
* **Root**: The topmost node in a tree is known as the root node. It has no parent and serves as the starting point for all nodes in the tree.
* **Parent Node**: A node that has one or more child nodes. In a binary tree, each node can have at most two children.
* **Child Node**: A node that is a descendant of another node (its parent).
* **Leaf Node**: A node that does not have any children or both children are null.
* **Internal Node**: A node that has at least one child. This includes all nodes except the **leaf**nodes.
* **Depth of a Node**: The number of edges from a specific node to the root node. The depth of the **root**node is zero.
* **Height of a Binary Tree**: The number of nodes from the deepest leaf node to the root node.

**Properties of Binary Tree**

* The maximum number of nodes at level **L** of a binary tree is**2L**
* The maximum number of nodes in a binary tree of height **H** is **2H – 1**
* Total number of leaf nodes in a binary tree = total number of nodes with 2 children + 1
* In a Binary Tree with **N** nodes, the minimum possible height or the minimum number of levels is **Log2(N+1)**
* A Binary Tree with**L** leaves has at least**| Log2L |+ 1** levels

## Types of Binary Tree

Binary Tree can be classified into multiples types based on multiple factors:

* **On the basis of Number of Children**
  + [Full Binary Tree](https://www.geeksforgeeks.org/full-binary-tree/)
  + [Degenerate Binary Tree](https://www.geeksforgeeks.org/introduction-to-degenerate-binary-tree/)
  + [Skewed Binary Trees](https://www.geeksforgeeks.org/skewed-binary-tree/)
* **On the basis of Completion of Levels**
  + [Complete Binary Tree](https://www.geeksforgeeks.org/complete-binary-tree/)
  + [Perfect Binary Tree](https://www.geeksforgeeks.org/perfect-binary-tree/)
  + [Balanced Binary Tree](https://www.geeksforgeeks.org/balanced-binary-tree/)
* **On the basis of Node Values:**
  + [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-data-structure/)
  + [AVL Tree](https://www.geeksforgeeks.org/introduction-to-avl-tree/)
  + [Red Black Tree](https://www.geeksforgeeks.org/introduction-to-red-black-tree/)
  + [B Tree](https://www.geeksforgeeks.org/introduction-of-b-tree-2/)
  + [B+ Tree](https://www.geeksforgeeks.org/introduction-of-b-tree/)
  + [Segment Tree](https://www.geeksforgeeks.org/segment-tree-data-structure/)

## Binary Search Tree

A **Binary Search Tree (BST)** is a type of binary tree data structure in which each node contains a unique key and satisfies a specific ordering property:

* All nodes in the left subtree of a node contain values **strictly less** than the node’s value.
* All nodes in the right subtree of a node contain values **strictly greater** than the node’s value.

This structure enables efficient operations for searching, insertion, and deletion of elements, especially when the tree remains balanced.

**Key Characteristics of a BST:**

* **Hierarchical Structure**: A BST is composed of nodes, each having up to two children, forming a tree-like hierarchy with a single root node at the top.
* **Ordering Property**: For every node in the BST, all values in the left subtree are smaller, and all values in the right subtree are larger than the node’s value. This rule holds recursively for all subtrees.
* **Efficient Operations**: In a balanced BST, operations like search, insertion, and deletion can be performed in **O(log n)** time. In the worst-case (unbalanced), these degrade to **O(n)**. With self-balancing BSTs like AVL and Red Black Trees, we can ensure the worst case as O(Log n).
* **Recursive Nature**: Each left or right subtree of a node in a BST is itself a BST, allowing recursive algorithms to naturally process the tree.
* **Practical Applications**: BSTs are widely used in database indexing, symbol tables, range queries, and are foundational for advanced structures like AVL trees, Red-Black trees. In problem solving, BSTs are used in problems where we need to maintain sorted stream of data.

## Balanced Binary Tree

A binary tree is balanced if the height of the tree is **O(Log n)** where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root-to-leaf path is the same and that there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.

* A single node is always balanced. It is also referred to as a height-balanced binary tree.
* An empty tree (Root = Null) is also always considered as balanced.

[How to Check if a Binary Tree is balanced?](https://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/)

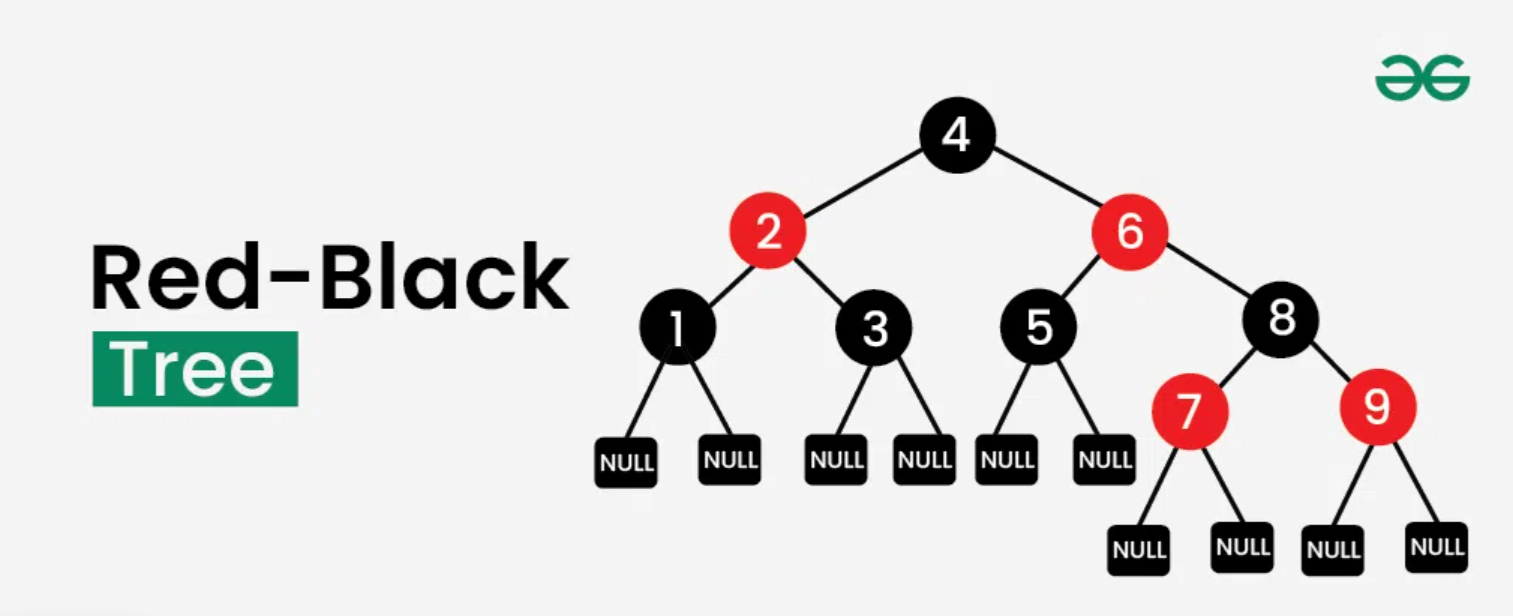
To check if a Binary tree is balanced we need to check three conditions :

1. The absolute difference between heights of left and right subtrees at any node should be less than 1.
2. For each node, its left subtree should be a balanced binary tree.
3. For each node, its right subtree should be a balanced binary tree.

We can solve this problem in O(n) time.

## Red-Black Tree

**Binary search trees** are a fundamental[data structure,](https://www.geeksforgeeks.org/data-structures) but their performance can suffer if the tree becomes unbalanced.**Red Black Trees**are a type of**balanced binary search tree** that use a set of rules to maintain balance, ensuring logarithmic time complexity for operations like**insertion, deletion, and searching**, regardless of the initial shape of the tree. **Red Black Trees** are self-balancing, using a simple color-coding scheme to adjust the tree after each modification.



A **Red-Black Tree**is a self-balancing[binary search tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion) where each node has an additional attribute: a color, which can be either **red** or **black**. The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.

**Properties of Red-Black Trees**

A[Red-Black Tree](https://www.geeksforgeeks.org/introduction-to-red-black-tree) have the following properties:

1. **Node Color**: Each node is either red or **black**.
2. **Root Property**: The root of the tree is always **black**.
3. **Red Property**: Red nodes cannot have red children (no two consecutive red nodes on any path).
4. **Black Property**: Every path from a node to its descendant null nodes (leaves) has the same number of **black** nodes.
5. **Leaf Property**: All leaves (NIL nodes) are **black**.

These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree's balance and efficient performance.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)**time where h is the height of the [BST](https://www.geeksforgeeks.org/binary-search-tree-data-structure). The cost of these operations may become**O(n)** for a skewed[Binary tree.](https://www.geeksforgeeks.org/binary-tree-data-structure) If we make sure that the height of the tree remains **O(log n)**after every insertion and deletion, then we can guarantee an upper bound of **O(log n)** for all these operations. The height of a Red-Black tree is always **O(log n)** where n is the number of nodes in the tree.

| **Sr. No.** | **Algorithm** | **Time Complexity** |
| --- | --- | --- |
| 1. | Search | O(log n) |
| 2. | Insert | O(log n) |
| 3. | Delete | O(log n) |

## Trie (Prefix Tree)

The**trie data structure**, also known as a**prefix tree**, is a tree-like data structure used for efficient retrieval of key-value pairs. It is commonly used for implementing dictionaries and autocomplete features, making it a fundamental component in many search algorithms.

**Trie data structure** is defined as a Tree based data structure that is used for storing a collection of strings and performing efficient search, insert, delete, prefix search and sorted-traversal-of-all operations on them. The word Trie is derived from re**TRIE**val, which means finding something or obtaining it.

**Trie data structure** follows a property that If two strings have a common prefix then they will have the same ancestor in the trie. This particular property allows to find all words with a given prefix.

A **Trie data structure** is used for **storing**and **retrieval**of data and the same operations could be done using another data structure which is **Hash Table**but Trie data structure can perform these operations more efficiently than a Hash Table. Moreover, Trie has its own advantage over the Hash table. A Trie data structure can be used for **prefix-based** searching and a sorted traversal of all words. So a Trie has advantages of both hash table and self balancing binary search trees. However the main issue with Trie is extra memory space required to store words and the space may become huge for long list of words and/or for long words.

**Properties of a Trie Data Structure**

Below are some important properties of the Trie data structure:

* Each Trie has an empty root node, with links (or references) to other nodes
* Each node of a Trie represents a string and each edge represents a character.
* Every node consists of hashmapsor**an array of pointers**, with each index representing a character and a flag to indicate if any string ends at the current node.
* Trie data structure can contain any number of characters including **alphabets**, **numbers**, and **special characters**.
* Each path from the root to any node represents a word or string.

## Heap

A **Heap** is a special tree-based data structure with the following properties:

* It is a **complete binary tree** (all levels are fully filled except possibly the last, which is filled from left to right).
* It satisfies either the **max-heap property** (every parent node is greater than or equal to its children) or the **min-heap property** (every parent node is less than or equal to its children).

**Note:** This definition applies specifically to **binary heaps**. Other types of heaps (like Fibonacci heaps or binomial heaps) may not be complete binary trees but still maintain the heap property in their own way. So if you mean **binary heap**, your original statement is correct and standard.

**Max-Heap**

The value of the root node must be the greatest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

**Min-Heap**

The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

**Properties of Heap:**

* The minimum or maximum element is always at the root of the heap, allowing constant-time access.
* The relationship between a parent node at index **'i'** and its children is given by the formulas: left child at index**2i+1** and right child at index**2i+2**for 0-based indexing of node numbers.
* As the tree is complete binary, all levels are filled except possibly the last level. And the last level is filled from left to right.
* When we insert an item, we insert it at the last available slot and then rearrange the nodes so that the heap property is maintained.
* When we remove an item, we swap root with the last node to make sure either the max or min item is removed. Then we rearrange the remaining nodes to ensure heap property (max or min)

**Operations Supported by Heap:**

Operations supported by **min - heap** and **max - heap** are same. The difference is just that min-heap contains minimum element at root of the tree and max - heap contains maximum element at the root of the tree.

**Heapify:**It is the process to rearrange the elements to maintain the property of heap data structure. It is done when root is removed (we replace root with the last node and then call heapify to ensure that heap property is maintained) or heap is built (we call heapify from the last internal node to root) to make sure that the heap property is maintained. This operation also takes **O(log n)** time.

**Insertion**: When a new element is inserted into the heap, it can disrupt the heap's properties. To restore and maintain the heap structure, a heapify operation is performed. This operation ensures the heap properties are preserved and has a time complexity of **O(log n)**.

* First increase the heap size by 1, so that it can store the new element.
* Insert the new element at the end of the Heap.
* This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify**this newly inserted element following a bottom-up approach.

**Deletion**:

If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.

Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.

**It takes O(log n) time.**

**getMax (For max-heap) or getMin (For min-heap):**

It finds the maximum element or minimum element for **max-heap** and **min-heap** respectively and as we know minimum and maximum elements will always be the root node itself for min-heap and max-heap respectively. It takes **O(1)** time.

**removeMin or removeMax:**

This operation returns and deletes the maximum element and minimum element from the max-heap and min-heap respectively. In short, it deletes the root element of the heap binary tree.

### Implementation of Heap Data Structure

The following code shows the implementation of a **max-heap**.

**maxHeapify**is the function responsible for restoring the property of the Max Heap. It arranges the node **i**, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, **arr[]** representing the complete binary tree. The left and the right child of **ith** node are in indices **2\*i+1** and **2\*i+2**.
2. We set the index of the current element, **i**, as the ‘MAXIMUM’.
3. If **arr[2 \* i + 1] > arr[i]**, i.e., the left child is larger than the current value, it is set as ‘MAXIMUM’.
4. Similarly if **arr[2 \* i + 2] > arr[i]**, i.e., the right child is larger than the current value, it is set as ‘MAXIMUM’.
5. Swap the ‘MAXIMUM’ with the current element.
6. Repeat steps **2 to 5** till the property of the heap is restored.

## Fibonacci Heap

1. A Fibonacci heap is a data structure used for implementing priority queues. It is a type of heap data structure, but with several improvements over the traditional binary heap and binomial heap data structures.
2. The key advantage of a Fibonacci heap over other heap data structures is its fast amortized running time for operations such as insert, merge and extract-min, making it one of the most efficient data structures for these operations. The running time of these operations in a Fibonacci heap is O(1) for insert, O(log n) for extract-min and O(1) amortized for merge.
3. A Fibonacci heap is a collection of trees, where each tree is a heap-ordered multi-tree, meaning that each tree has a single root node with its children arranged in a heap-ordered manner. The trees in a Fibonacci heap are organized in such a way that the root node with the smallest key is always at the front of the list of trees.
4. In a Fibonacci heap, when a new element is inserted, it is added as a singleton tree. When two heaps are merged, the root list of one heap is simply appended to the root list of the other heap. When the extract-min operation is performed, the tree with the minimum root node is removed from the root list and its children are added to the root list.
5. One unique feature of a Fibonacci heap is the use of lazy consolidation, which is a technique for improving the efficiency of the merge operation. In lazy consolidation, the merging of trees is postponed until it is necessary, rather than performed immediately. This allows for the merging of trees to be performed more efficiently in batches, rather than one at a time.

In summary, a Fibonacci heap is a highly efficient data structure for implementing priority queues, with fast amortized running times for operations such as insert, merge and extract-min. Its use of lazy consolidation and its multi-tree structure make it a superior alternative to traditional binary and binomial heaps in many applications.

**1) Find Min: Θ(1) [Same as Binary but not Binomial since binomial has o(log n)]  
2) Delete Min: O(Log n) [Θ(Log n) in both Binary and Binomial]  
3) Insert: Θ(1) [Θ(Log n) in Binary and Θ(1) in Binomial]  
4) Decrease-Key: Θ(1) [Θ(Log n) in both Binary and Binomial]  
5) Merge: Θ(1) [Θ(m Log n) or Θ(m+n) in Binary and  
 Θ(Log n) in Binomial]**

# Graph Algorithms

## Graphs

**Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( **V**) and a set of edges( **E**). The graph is denoted by **G(V, E).**

## Breadth First Search

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent nodes. Once all adjacent are visited, then their adjacent are traversed.

BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level.

Popular graph algorithms like Dijkstra’s shortest path, Kahn’s Algorithm, and Prim’s algorithm are based on BFS.

BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

**Complexity: O(|V| + |E|)**

## Depth First Search

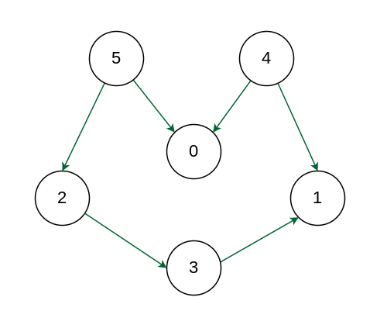
In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

**Complexity: O(|V| + |E|)**

## Topological Sorting

Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.

**Note:** Topological Sorting for a graph is not possible if the graph is not a **DAG**.



Example

**Output:** 5 4 2 3 1 0  
**Explanation:** The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).  A topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. Another topological sorting of the following graph is “4 5 2 3 1 0”.

**Algorithm for Topological Sorting using DFS:**

Here’s a step-by-step algorithm for topological sorting using Depth First Search (DFS):

* Create a graph with **n** vertices and **m**-directed edges.
* Initialize a stack and a visited array of size **n**.
* For each unvisited vertex in the graph, do the following:
  + Call the DFS function with the vertex as the parameter.
  + In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.
  + Once all the neighbors have been visited, push the vertex onto the stack.
* After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.
* The resulting list is the topologically sorted order of the graph.

## Dijkstra’s Algorithm

[*Dijkstra’s algorithm*](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) *is a popular algorithm for solving single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist* ***Edsger W. Dijkstra*** *in 1956.*

**Algorithm for Dijkstra's Algorithm**

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

*The problem with negative weights arises from the fact that Dijkstra's algorithm assumes that once a node is added to the set of visited nodes, its distance is finalized and will not change. However, in the presence of negative weights, this assumption can lead to incorrect results.*

Dijkstra's algorithm has a time complexity of **O(V^2)** for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.

## Bellman–Ford Algorithm

**O(V\*E) Time and O(V) Space**

The Bellman-Ford algorithm helps find the shortest path from one starting point to all other points in a graph, even if some paths have negative weights. It's useful for network routing problems. In this article, we will learn about the Bellman-Ford algorithm and how to implement it in C.

The Bellman-Ford algorithm finds the shortest path from one starting point to all other points in a graph. It works on graphs with weights and without weights. Like Dijkstra's algorithm, it guarantees finding the shortest path, but it's slower. Unlike Dijkstra's, it can handle graphs with negative weights. However, it can't find a shortest path if there are any negative cycles (where you can keep subtracting distance forever).

**Principle of Bellman Ford Algorithm**

Bellman-Ford is based on **Principle of Relaxation**. It starts by assuming that the shortest distance to all vertices from the source vertex is infinity. Then, through the series of iterations, it update all these distances by relaxing the edges. Means finding shorter paths whenever possible.

**Detecting Negative Cycles**

* For a graph with N vertices, all the edges should be relaxed N-1 times to find the single source shortest path.
* In order to detect whether a[negative cycle](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/) exist or not, relax all the edges one more time and if the shortest distance for any node reduces then we can say that negative cycle exist there.

**Why relaxing of edges is N- times?**

* In worst case, a shortest path between two vertices can have at most N- edges. By relaxing edges N- times the algorithm optimizes the distance estimates for all vertices, assuming the absence of negative weight cycles reachable from source vertex.

**Algorithm**

1. *Initialize the distance from the source to all vertices as infinite.*
2. *Set the distance to the source itself as 0.*
3. *Set the predecessor of all vertices as NULL.*
4. *For each vertex, apply relaxation for all its edges.*
5. *Repeat the relaxation process V-1 times, where V is the number of vertices in the graph.*
6. *Relaxation is the process of updating the shortest path found so far by checking if a shorter path can be obtained by going through the vertex under consideration.*
7. *Check for negative-weight cycles by relaxing all edges one more time.*
8. *If we can find a shorter path, then there is a negative cycle.*
9. *Return the shortest distance array and the predecessor array.*

## Floyd Warshall Algorithm

The Floyd Warshall Algorithm is an all-pair shortest path algorithm that uses Dynamic Programming to find the shortest distances between every pair of vertices in a graph, unlike Dijkstra and Bellman-Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs and can handle graphs with both positive and negative weight edges. **Algorithm**

* *Initialize the solution matrix the same as the input graph matrix as a first step.*
* *Then update the solution matrix by considering all vertices as an intermediate vertex.*
* *The idea is to pick all vertices one by one and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.*
* *When we pick vertex number****k****as an intermediate vertex, we already have considered vertices****{0, 1, 2, .. k-1}****as intermediate vertices.*
* *For every pair****(i, j)****of the source and destination vertices respectively, there are two possible cases.*
  + ***k****is not an intermediate vertex in the shortest path from****i****to****j****. We keep the value of****dist[i][j]****as it is.*
  + ***k****is an intermediate vertex in the shortest path from****i****to****j****. We update the value of****dist[i][j]****as****dist[i][k] + dist[k][j],****if****dist[i][j] > dist[i][k] + dist[k][j]***

**Time Complexity:**O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V

**Auxiliary Space:**O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

## Maximum Flow Problem

The maximum flow problem is a classic optimization problem in network theory, where the goal is to determine the maximum possible flow from a **source**node to a **sink**node in a flow network. A flow network is represented as a directed graph with each edge having a capacity, which is the maximum amount of flow that can pass through it. The problem finds applications in various fields such as computer networks, logistics, and transportation systems.

**Algorithm to Find Maximum Flow in Python**

The most commonly used algorithm to solve the maximum flow problem is the [Ford-Fulkerson method](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-in-python/). This method involves finding augmenting paths in the network and increasing the flow along these paths until no more augmenting paths can be found.

**Steps of the Ford-Fulkerson Algorithm:**

1. **Initialize flow**: Start with zero flow in all edges.
2. **Find augmenting path**: Use Depth-First Search ([DFS](https://www.geeksforgeeks.org/python-program-for-depth-first-search-or-dfs-for-a-graph/)) or Breadth-First Search ([BFS](https://www.geeksforgeeks.org/python-program-for-breadth-first-search-or-bfs-for-a-graph/)) to find a path from the source to the sink such that the capacity along the path is greater than the current flow.
3. **Update flow**: Increase the flow along the found path by the minimum residual capacity (bottleneck capacity) of the path.
4. **Repeat**: Continue finding augmenting paths and updating flows until no more augmenting paths can be found.

The **Edmonds-Karp algorithm** is an implementation of the **Ford-Fulkerson method** using BFS to find the augmenting paths, which ensures the algorithm runs in polynomial time.

## 0-1 BFS (Shortest Path in a Binary Weight Graph)

Given an undirected graph where every edge has a weight as either 0 or 1. The task is to find the **shortest path** from the source vertex to all other vertices in the graph.

*The idea is to adapt BFS to efficiently handle graphs with binary weights (0 or 1) by using a deque instead of a queue. When we encounter an edge with weight 0, we add its destination to the front of the deque because it doesn't increase the distance. When we encounter an edge with weight 1, we add its destination to the back of the deque because it increases the distance by one unit.*

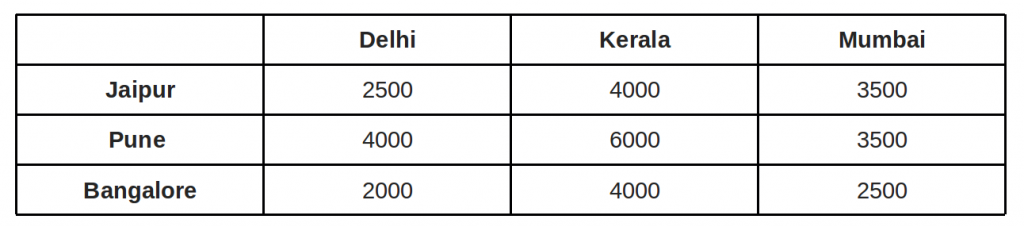
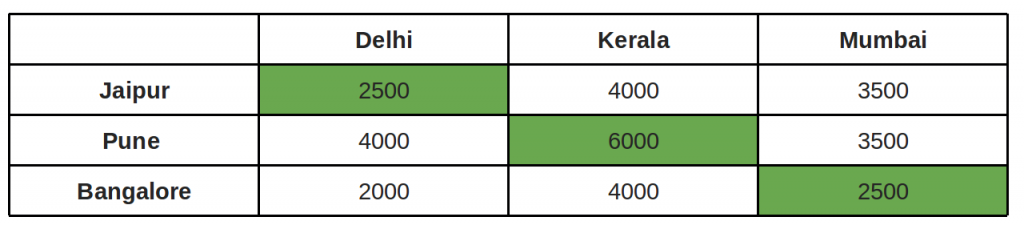
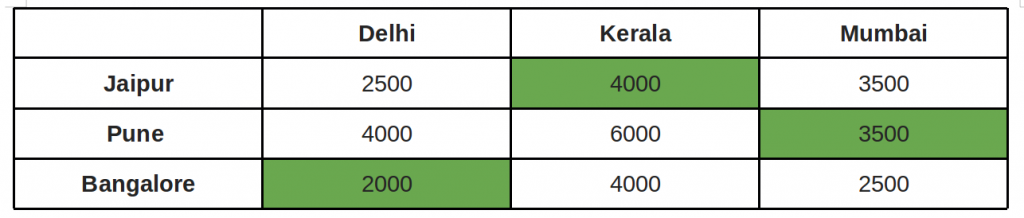
Step by step approach:

1. Initialize distances array with infinity for all vertices except source.
2. Use a deque to process vertices in order of increasing distance.
3. For each vertex, examine all adjacent vertices. If the new distance of vertex is less than current distance, then
   * If edge weight is 0, add adjacent vertex to front of deque (priority).
   * If edge weight is 1, add adjacent vertex to back of deque.

### Using Deque - O(V + E) time and O(V + E) space

# Other Algorithms

## Hungarian Algorithm for Assignment Problem

Let there be n agents and n tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized. **Example:** You work as a manager for a chip manufacturer, and you currently have 3 people on the road meeting clients. Your salespeople are in Jaipur, Pune and Bangalore, and you want them to fly to three other cities: Delhi, Mumbai and Kerala. The table below shows the cost of airline tickets in INR between the cities: The question: where would you send each of your salespeople in order to minimize fair? Possible assignment: Cost = 11000 INR Other Possible assignment: Cost = **9500** INR and this is the best of the **3!** possible assignments. **Brute force solution** is to consider every possible assignment implies a complexity of **Ω(n!)**. The **Hungarian algorithm, aka Munkres assignment algorithm**, utilizes the following theorem for polynomial runtime complexity (**worst case O(n3)**) and guaranteed optimality: *If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix.* We reduce our original weight matrix to contain zeros, by using the above theorem. We try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is **zero**. **Core of the algorithm (assuming square matrix):**

1. For each row of the matrix, find the smallest element and subtract it from every element in its row.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. Do the same (as step 1) for all columns.
2. Cover all zeros in the matrix using minimum number of horizontal and vertical lines.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. *Test for Optimality:* If the minimum number of covering lines is n, an optimal assignment is possible and we are finished. Else if lines are lesser than n, we haven’t found the optimal assignment, and must proceed to step 5.
2. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

## Knuth-Morris-Pratt

The Knuth-Morris-Pratt (KMP) is an efficient string-matching algorithm developed by Donald Knuth, Vaughan Pratt, and James H. Morris in 1977. It is used for finding a specific pattern in the given string.

The [Knuth-Morris-Pratt (KMP) algorithm](https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/) is a popular and efficient method to find the smaller string (termed as the "**pattern**") inside a larger string (termed as the "**text**"). It speeds up the search process by avoiding unnecessary comparisons by precomputing the . The KMP algorithm works in two main steps:

* **Preprocessing the Pattern**: Before searching, KMP creates an array (called the "LPS" array) based on the **pattern** that helps determine how much of the **pattern**has already been matched.
* **Searching the Text**: Using the LPS array, KMP quickly skips over parts of the **text**where it's certain the pattern can't match, making the search more efficient.

**Create the LPS Array**

The [LPS array](https://www.geeksforgeeks.org/longest-prefix-also-suffix/) tells us how much of the pattern can be reused when a mismatch occurs. Here’s how we create the LPS array for the pattern P = "ABABCABAB":

| **Index(i)** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Pattern** | A | B | A | B | C | A | B | A | B |
| **LPS Value** | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 |

The LPS array helps the algorithm skip unnecessary comparisons by showing how much of the pattern can be reused after a mismatch.

**Match the Pattern Against the Text**

Now, using the LPS array, we compare the pattern with the text:

* Start at the beginning of both the text and the pattern.
* Compare characters one by one.
* If they match, move both pointers (one in the text and one in the pattern) forward.
* If they don't match, use the LPS array to decide how far to skip ahead in the pattern without re-checking parts of the text we've already looked at.

## Backtracking

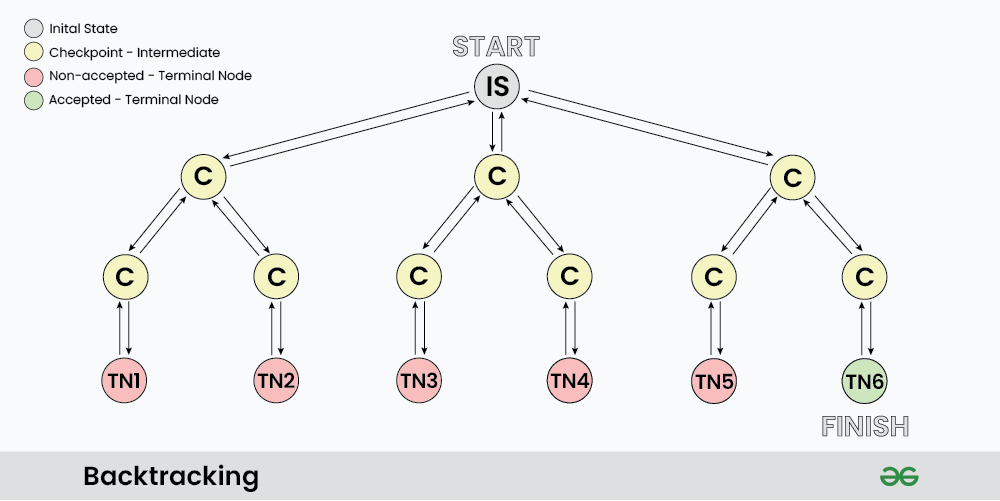
Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**.

* It is commonly used in situations where we need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.
* When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.
* We use backtracking when we need to explore all possible paths (or permutations). Instead of going through every path, we backtrack to avoid unnecessary work whenever we are sure that no path from here would lead to a valid solution.

**Types of Backtracking Problems**

Problems associated with backtracking can be categorized into 3 categories:

* **Decision Problems:** Here, we search for a feasible solution.
* **Optimization Problems:**For this type, we search for the best solution.
* **Enumeration Problems:** We find set of all possible feasible solutions to the problems of this type.



**Determining Backtracking Problems**

Generally every constraint satisfaction problem can be solved using backtracking but, Is it optimal to use backtracking every time? Turns out **NO**, there are a vast number of problem that can be solved using [Greedy](https://www.geeksforgeeks.org/dsa/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/) or [Dynamic programming](https://www.geeksforgeeks.org/competitive-programming/dynamic-programming/) in logarithmic or polynomial time complexity which is far better than exponential complexity of Backtracking. However many problems still exists that can only be solved using Backtracking.

*Technically, for backtracking problems:*

* *The algorithm builds a solution by exploring all possible paths created by the choices in the problem, this solution begins with an empty set* ***S={}***
* *Each choice creates a new sub-tree '****s'*** *which we add into are set.*
* *Now there exist two cases:*
  + ***S+s is valid set***
  + ***S+s is not valid set***
* *In case the set is valid then we further make choices and repeat the process until a solution is found, otherwise we backtrack our decision of including '****s'*** *and explore other paths until a solution is found or all the possible paths are exhausted.*

***Applications of Backtracking***

* *Creating smart bots to play Board Games such as Chess.*
* *Solving mazes and puzzles such as N-Queen problem.*
* *Network Routing and Congestion Control.*
* *Decryption*
* *Text Justification*

### How Backtracking is different from Recursion?

Recursion and Backtracking are related concepts in computer science and programming, but they are not the same thing. Let's explore the key differences between them:

| **Recursion** | **Backtracking** |
| --- | --- |
| Recursion does not always need backtracking | Backtracking always uses recursion to solve problems |
| Solving problems by breaking them into smaller, similar subproblems and solving them recursively. | Solving problems with multiple choices and exploring options systematically, backtracking when needed. |
| Controlled by function calls and call stack. | Managed explicitly with loops and state. |
| **Applications of Recursion:** Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search. | **Application of Backtracking:**N Queen problem, Rat in a Maze problem, Knight’s Tour Problem, Sudoku solver, and Graph coloring problems. |