Data Structures And Algorithms

Data Structures

Data Structure is a way to store and organize data so that it can be used efficiently.

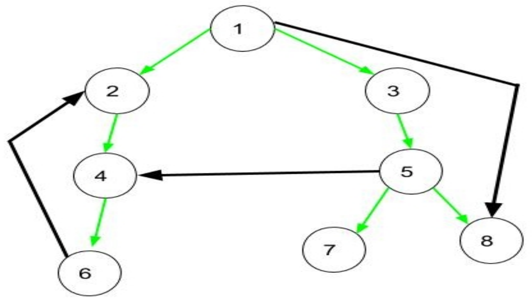
Tree : *A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels. In the Tree data structure, the topmost node is known as a root node.*

Ancestor : *A node that is connected to all lower-level nodes is called an "ancestor".(like Parent)*

Descendant : *The connected lower-level nodes are "descendants" of the ancestor node.(like Child)*

Edge Classsifications :

Tree Edge: *It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.*
Forward Edge: *It is an edge (u, v) such that v is descendant but not part of the DFS tree. Edge from 1 to 8 is a forward edge.*
Back edge: *It is an edge (u, v) such that v is ancestor of node u but not part of DFS tree. Edge from 6 to 2 is a back edge. Presence of back edge indicates a cycle in directed graph.*
Cross Edge: *It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them. Edge from node 5 to 4 is cross edge.*



Forest : *A collection of disjoint trees is called a forest.*

Data Structures	**Definition**	**Time Complexity**			**Space Complexity**	**Type**	**Adv./Disadvantages**	**Real Life Applications**
		Insert	**Delete**	**Search**				
Array	It is a data structure for storing more than one data item that has a similar data type. The items of an array are allocated at adjacent memory locations.	**O(n)**	**O(n)**	**o(n)**	**o(n)**	*Linear*	*i. Accessing an element is very easy by using the index number.* *ii. 2D Array is used to represent matrices.* *iii. ils size is always fixed.* *iv . only one type of data is stored.*	*i. we wish to store the contacts on our phone, then the software will simply place all our contacts in an array.*
Stack	Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).	**O(1)**	**O(1)**	**O(n)**	**O(n)**	*Linear*	*i. It allows you to control how memory is allocated and deallocated.* *ii. Random access of elements is impossible in stacks.*	*i.A stack of plates in a cupboard*
Queue	A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).	**O(1)**	**O(1)**	**O(n)**	**O(n)**	*Linear*	*i. Data queues are fast and optimised.* *ii. Inserting and removing elements from the middle is complex.*	*i. A queue of people at ticket-window*

Singly Linked List	A linked list is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a node. A node is a collection of two sub-elements or parts. A data part that stores the element and a next part that stores the link to the next node. A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).	**O(1)**	**O(1)**	**O(n)**	**O(n)**	*Linear*	*i. it is very easier for the accessibility of a node in the forward direction. ii. The insertion and deletion of a node are very easy. iii. it is very easier for the accessibility of a node in the forward direction. the insertion and deletion of a node are very easy. iv. the Accessing of a node is very time-consuming.*	*i. Undo button of any application like Microsoft Word, Paint, etc*
Doubly Linked List	A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.	**O(1)**	**O(1)**	**O(n)**	**O(n)**	*Linear*	*i. It can allocate or reallocate memory easily during its execution. ii. Reversing the doubly linked list is very easy. iii. It uses extra memory when compared to the array and singly linked list. iv. Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.*	*i. A music player which has next and previous buttons.*
Priority Queue	A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue	**O(log n)**	**O(log n)**	**O(log n)**	**O(n)**	*Linear/Sequential*	*i. Easy to implement ii. Dynamic memory allocation enabled iii. Memory overhead*	*i. A hospital queue where the patient with the most critical situation would be the first in the queue.*
Binary Heap	A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Let's say if X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree.	**O(log n)**	**O(log n)**	**O(n)**	**O(log n)**	*Non-linear*	*i. It helps to find greatest and minimum number. ii.It helps to find greatest and minimum number*	i. A hospital queue where the patient with the most critical situation would be the first in the queue.
Binary Tree	A binary tree is a tree data structure in which each node has at most two children, where each node contains following parts. 1.Data 2.Pointer to left child 3.Pointer to right child	**O(n)**	**o(n)**	**O(n)**	**O(n)**	*Non-linear*	*i. It saves space because no pointers are stored. ii. Accessing any node in the tree requires sequentially processing all nodes that appear before it in the node list.*	*i. Routing tables. ii. Data compression.*

Binary Search Tree	a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than or equal to the data of the root and The data of all the nodes in the right subtree of the root node should be greater than the data of the root.	**O(log n)**	**o(log n)**	**O(log n)**	**O(n)**	*Non-linear*	*i. It speed up the insertion and deletion operations as compare to that in array and linked list. ii. Shape of the tree depends upon order of insertion and it can be degenerated.*	*i. Maintain sorted stream of data. For example, suppose we are getting online orders placed and we want to maintain the live data (in RAM) in sorted order of prices.*
B-Tree	B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree. It is also known as a height-balanced m-way tree.	**O(Logn)**	**O(Logn)**	**O(Logn)**	**O(n)**	*Non-linear*		*i. Databases and file systems*
AVL Tree	AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.	**O(Logn)**	**O(Logn)**	**O(Logn)**	**O(n)**	*Non-linear*	*i. Better search times for keys. ii. Longer running times for the insert and remove operations*	*i.Database Application*
Hashing	Hashing is a technique or process of mapping keys, values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.	**O(1)**	**O(1)**	**O(1)**	**O(n)**	*Linear/Sequential*	*i.Hash tables turn out to be more efficient than search trees or any other table lookup structure. ii. Hash tables become quite inefficient when there are many collisions. iii. Hash collisions are practically unavoidable.*	*i. Password verification*
Hashing With Chaining	Chaining is a technique used for avoiding collisions in hash tables.In the chaining approach, the hash table is an array of linked lists i.e., each index has its own linked list. All key-value pairs mapping to the same index will be stored in the linked list of that index.(Complexity is O (1) if α is of O(1))	**O(1)**	**O(1)**	**O(1)**	**O(n)**	-	*i. Hash table never fills up, we can always add more elements to the chain. ii. Wastage of Space (Some Parts of hash table are never used. iii.If the chain becomes long, then search time can become O(n) in the worst case.*	-
Open Addressing	Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). α =n/m where n = no. of elements, m = no. of slots	**O(1/(1 - α))**	**O(1/(1 - α))**	**O(1/(1 - α))**	**O(n)**	-	-	-
Perfect Hashing	Perfect hashing is defined as a model of hashing in which any set of n elements can be stored in a hash table of equal size and can have lookups performed in constant time.	**o(1)**	**O(1)**	**O(1)**	**O(n)**	-	-	-

Van Emde Boas Tree	VEBT is a tree data structure which implements an associative array . It performs all operations (insert, delete, lookup, maximum, minimum, successor and predecessor) in O(log log u) time, where u is the size of the universe for storing n elements in it.	**O(log log u)**	**O(log log u)**	**O(log log u)**	**O(u)**	-	*i. It works faster for all operations. ii. It works with O(1) time-complexity for minimum and maximum query.*	-

Divide & Conquer

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

In-order Traversal : *In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.*

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order

Pre-order Traversal : *In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.*

Post-order Traversal : *In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.*

Searching Algorithms

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	**Type**
		Average case	**Worst case**					
Linear Search	Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.	**O(n)**	**O(n)**	**O(1)**	i.When a key element matches the first element in the array, then linear search algorithm is best case because executing time of linear search algorithm is 0 (n), where n is the number of elements in an array. ii. The list doesn't have to sort. iii. The drawback of a linear search is the fact that its time consuming for the enormous arrays.	i. For small no. of data	i. Search for a number in Phonebook.	Comparison

Binary Search	A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search.	**O(log n)**	**O(log n)**	**O(1)**	i. It is more efficient in the case of large-size data sets. ii.The pre-condition for the binary search is that the elements must be arranged in a sorted order.	i. For large size data.	i. Find a page Number in a book : we want to go to page 345 in a book. We don't visit every page from 1 to 345. We open any random page on the book and check it's page number. If the current page number is greater than 345, then 345 is on the left side of the current page. If the current page number is smaller than 345, then 345 is on the right side of the current page. We keep repeating the process until we reach page 345.	Comparison

Sorting Algorithms

A sorting algorithm is an algorithm that puts elements of a list into an order.

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	**Type**
		Average case	**Worst case**					
Bubble Sort	Bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.	**O(n^2)**	**O(n^2)**	**O(1)**	i. Too slow ii. Impractical for most problem	i. Almost sorted array with some out of order elements	i. It is used in programming TV to sort channels based on audience viewing time.	Comparison
Insertion Sort	Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.	**O(n^2)**	**O(n^2)**	**O(1)**	i. Worst Case if i/p is reverse sorted ii. In-efficient for large data iii. Better than Selection or Bubble (in practice) iv. Gets penalized if comparison or copying is slow.	i. Small arrays ii. Partially or almost sorted arrays	i. Tailors arrange customers' shirts in their wardrobe, according to size. So they insert a new shirt at the proper position, for that, they shift existing shirts until they find the proper place.	Comparison
Selection Sort	This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.	**O(n^2)**	**O(n^2)**	**O(1)**	i. Performs better than Bubble Sort ii. Its worst case is extremely poor.	i. Small arrays	i. It can be useful when memory write is a costly operation.	Comparison
Merge Sort	Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.	**O(n*log n)**	**O(n*log n)**	**O(n)**	i. Requires O(n) extra space ii. Only O(nlgn) stable sort iii. Quick Sort performs better in case of arrays.	i. Linked Lists	i. The e-commerce application Have you ever noticed on any e-commerce website, they have this section of "You might like", they have maintained an array for all the user accounts and then whichever has the least number of inversion with your array of choices, they start recommending what they have bought or they like.	Comparison

Quick Sort	Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot	**O(n*log n)**	**O(n^2)**	**O(n)**	i. Optimized version is used in many std. libraries ii. Fastest sorting alg but unbalanced partition can lead to very slow performance iii. Difficult to choose good pivot element	i. When avg. case performance matters more than WC performance ii. Arrays	i. Commercial Computing is used in various government and private organizations for the purpose of sorting various data like sorting files by name/date/price. ii. Sports scores are quickly organized	Comparison
Counting Sort	It sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.	**O(n+k)**	**O(n+k)**	**O(n+k)**	"i. Stable ii. Sorts in linear time if k=O(n) iii. Used as a sub-routine in Radix Sort	i. When k=O(n) ii. When sorting integers with limited range	i. When sort a list of bounded (e. g. less than some relatively small value) integers in linear time.	Non-comparison
Heap Sort	Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.	**O(n*log n)**	**O(n*log n)**	**O(n)**	i. Slower than Quick and Merge Sort ii. Used where guarenteed O (nlgn) time is required (Quick's WC is O(n2)) and less memory is available (Merge uses O(n) extra space)	i. Large arrays ii. When you care more about WC performance rather than avg. case performance	i. It is applied to a sim card store where there are many customers in line. The customers who have to pay bills can be dealt with first because their work will take less time.	Comparison
Bucket Sort	Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm.	**O(n)**	**O(n^2)**	**O(n+k)**	i.Bucket sort allows each bucket to be processed independently. ii.The problem is that if the buckets are distributed incorrectly, you may wind up spending a lot of extra effort for no or very little gain. iii.Bucket sort's performance is determined by the number of buckets used.	i.when input is uniformly distributed over a range.	i. It is used for floating point values.	Non-comparison
Radix Sort	It sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.	**O(d(n+k))**	**O(d(n+k))**	**O(n+k)**	i. Uses Counting Sort as subroutine ii. Used to sort records that are keyed by multiple fields like date	i. When d = O(1) and k = O(n) ii. When Counting Sort works well on digits of numbers to be sorted.	i. It is used in parallel computing.	Non-comparison

Graph Algorithms

A Graph is a data structure consisting of vertices(nodes) and edges.

Connected Graph : *if there is a path between connecting every pair of nodes or vertices.*

Disconnected Graph : *if there is no path between any pair of nodes or vertices.*

Directed Graph : *A directed graph or diagraph is a graph in which all the edges are directed.*

Undirected Graph : *It is collection of vertices and edges that are connected together without any direction.*

Weighted Graph : *if some real number is assigned on every edge of a graph*

Unweighted Graph : *if the edges have no weight,*

Bi-partite Graph : *A bipartite graph is a graph whose vertices can be divided into two disjoint set S1 and S2 such that every edge connected a vertex in S1 to one in S2.*

Graph Representations : *By Adjacency Matrix and Adjacency List.*

Negative Cycle : *A negative cycle is one in which the overall sum of the cycle becomes negative.*

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	

		Average case	Worst case					
BFS	Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.	**O(V+E)**	**O(V+E)**	**O(|V|)**	i. BFS will never get trapped in a blind alley, which means unwanted nodes. ii. If there is more than one solution then it will find a solution with minimal steps. iii. If a solution is far away then it consumes time.	i. For searching vertices which are closer to the given source.	i. Social Networking Websites. ii. Broadcasting in Network.	
DFS	Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.	**O(V+E)**	**O(V+E)**	**O(|V|)**	i. Less time and space complexity rather than BFS. ii. The solution can be found out without much more search. iii. Not Guaranteed that it will give you a solution. iv. Determination of depth until the search has proceeded.	i.Decision tree	i. Path finding. ii. Solving puzzles with only one solution, such as mazes.	
Topological Sort	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.The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).	**O(V+E)**	**O(V+E)**	**O(|V|)**	i. To find a cycle in a graph. ii. Not useful for undirected graphs.	i.Directed Acyclic graph.	i. It is mainly used for scheduling jobs from the given dependencies among jobs.	
Dijkstra	shortest paths from the source to all vertices in the given graph.(it works for positive edge weight only)	**O(V lg V + E)**	**O(V lg V + E)**	**O(V^2)**	i. It has lower complexity than Bellmon ford. ii. Cannot used for negative edge weights.	i. For positive routes path.	i. Digital Mapping Services in Google Maps. ii. Telephone Network	
Bellmon Ford	shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.	**O(V E)**	**O(V E)**	**O(V)**	i. It can handle negative weights. ii. Its work slower than Dijkstra.	i. To Find a path form source to all destinations.	i. Used in Routing protocol. ii. Shortest path	
Floyd-Warshall Algorithm	It is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles	**O(V^3)**	**O(V^3)**	**O(|V^2|)**	i. A single execution of the algorithm is sufficient to find the lengths of the shortest paths between all pairs of vertices. ii. A single execution of the algorithm is sufficient to find the lengths of the shortest paths between all pairs of vertices. iii. It can find the shortest path only when there are no negative cycles. iv. It works slower than other algorithms designed to perform the same task.	i.Dense graphs(more edges).	i. Google Maps. ii. Social Network Analysis	
Johnson's algorithm	The problem is to find the shortest path between every pair of vertices in a given weighted directed graph and weight may be negative. Using Johnson's Algorithm, we can find all pairs shortest path in O (V^2 log V+VE) time. Johnson's Algorithm uses both Dijkstra's Algorithm and Bellman-Ford Algorithm.	**O(V^2 log V + VE)**	**O(V^3 + VE)**	**O(V^2)**	i. It works Faster than Floyd Warshall Algorithms. ii. It works for negative weight edge also.	i.Sparse graphs(few edges).	i. Google Maps. ii. Railway routes.	

Greedy Algorithms

A greedy algorithm repeatedly makes a locally best choice or decision, but ignores the effects of the future.

Properties Of Greedy Algorithm :

1) Greedy Choice Property : *A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.*

2) Optimal substructure : *Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.*

Spanning Tree : *A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G.A connected graph can contain more than one spanning tree*

Minimum Spanning Tree : *The minimum spanning tree is the tree whose sum of the edge weights is minimum. it consist only V-1 edges because if it has V edge the tree become cyclic.*

For finding Miniumum Spanning Tree : *It can be find by two algorithms namely* ***Prim's*** *and* ***Krsukal's algorithms.***

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	
		Average case	**Worst case**					
Prim's Algorithm	It starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered. if using array it takes O (v^2) time and using binary heap it takes O(E log V) time.	**O(E log V)**	**O(V^2)**	**O(|E| + |V|)**	i. Its complexity is better than Kruskal's algorithm. ii. It doesn't allow us much control over the chosen edges when multiple edges with the same weight occur. iii. It is harder to implement.	i. Dense graphs.	i. Network for roads and Rail tracks connecting all the cities. ii. Travelling Salesman Problem. iii. Designing a fiber-optic grid or ICs.	
Kruskal's Algorithm	It constructs an MST by taking the globally lowest-weight edge and contracting it.	**O(E log E)**	**O(E log E)**	**O(|E| + |V|)**	i. It is better to use regarding the easier implementation and the best control over the resulting MST	i. Sparse graphs.	i. LAN Networks ii. TV Network iii. A network of pipes for drinking water or natural gas.	

Randomized Algorithms

Algorithm that generates a random number r ∈ {1, ..., R} and makes decision based on r's value .On the same input on different executions, a randomized algorithm may

– Run a different number of steps

– Produce a different output

Classification Of Randomized Algorithms :

1) Monte Carlo - *runs in polynomial time always output is correct with high probability.*

2) Las Vegas - *runs in expected polynomial time output always correct.*

Algorithm	**Definition**	**Time Complexity**	**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	
Randomized Quick Sort	Implemment a quick sort using random pivot at each recursion we choose a pivot element randomly. it works in O(n*log n) expected time. since the output is always correct, it is Las Vegas randomized algorithm.	**O(n*log n)**	**O(log n)**	i.Its output is always correct.	-	i. Commercial Computing	

Frievald's Algorithm	It is a probabilistic randomized algorithm used to verify matrix multiplication. It works in time O(n^2) with high probability. In O(kn^2) time the algorithm can verify a matrix product with probability of failure less than 2-^-k. Since the output is not always correct, it is a Monte Carlo randomized algorithm.	**O(n^2)**	**O(n^2)**	i. It checks matrix multiplication faster. ii. It output is not always correct.	-	i. Mathematics : for faster matrix multiplicaton check	

Randomized Data Structures

Any probablistic data structure will rely on some form of probablity such as using randomness, hasing, and etc to reach an approximate solution.

Universal Hashing : *It refers to selecting a hash function at random from a family of hash functions with a certain mathematical property.*

Data Structures	**Definition**	**Time Complexity**			**space complexity**	**Best suited for**	**Adv./Disadvantages**	**Real Life Applications**
		Insert	**Delete**	**search**				
Randomized Skip list	A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list. The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly.	**O(log n)**	**O(log n)**	**O(log n)**	**O(n*log n)**	-	i. It is very simple to find a node in the list because it stores the nodes in sorted form. ii. The skip list is simple to implement as compared to the hash table and the binary search tree. iii. Reverse searching is not allowed. iv. The skip list searches the node much slower than the linked list.	i. Express and local subway lines : • Express line connects a few of the stations • Local line connects all stations • Links between lines at common stations

For Finding Roots

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	
		Average case	**Worst case**					
Newton's Method	Newton's method is a root finding method that uses linear approximation. In particular, we guess x1 a solution of the equation F(x)=0, compute the linear approximation of F(x) at x1 and then find the x-intercept of the linear approximation.where F(n) is the cost of calculating f(x)/f'(x)\, with n-digit precision.	**O((log n) F(n))**	**O((log n) F(n))**	**O((log n) F(n))**	i. As we go near to root, number of significant digits approximately with double each step. ii. It is slow convergence rate and thousands of iteration may happen around critical point.	-	i. for analysis of flow in water distribution networks.	

For Finding Multiplication

Algorithm	**Definition**	**Time Complexity**		**Space complexity**	**Adv./Disadavntages**	**Best Suited for**	**Real Life Applications**	
		Average case	**Worst case**					

Karatsuba Multiplication	The Karatsuba algorithm is a fast multiplication algorithm that uses a divide and conquer approach to multiply two numbers. The naive algorithm for multiplying two numbers has a running time of O (n^2)) while this algorithm has a running time of O(n^1. 585)	**O(n^1.585)**	**O(n^1.585)**	**O(n)**	i. It works better than Naive algorithm. ii. It is fast multiplication algorithm.	-	i. Multiplication of two numbers.	

Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Recursion : *The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.*

Recursion Tree : *In this method, we convert the recurrence into a tree and then we sum the costs of all the levels of the tree.*

Decision Tree : *Decision tree is the most powerful and popular tool for classification and prediction. A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes)*

Optimal Substructure : *A problem has an optimal substructure if its optimal solution can be constructed from the optimal solutions of its subproblems.*

Characteristics of Dynamic Programming :

1) Overlapping Subproblems

2) Optimal Substructure Property

Methods For Solving Dynamic Programming :

1) Top-Down Method : *In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called Memoization.*

2) Bottum-Up Method : *Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem "bottom-up" (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.*

Steps To Solve Dynamic Programming Problem :

1. define subproblems

2. guess (part of solution)

3. compute time/subproblem

4. recurse + memoize problems OR build DP table bottom-up

5. solve original problem: = combining subproblem solutions

Running Time For DP Problem :

Total Time = (time/subproblem) * no.of subproblems.

Reductions : *Convert your problem into a problem you already know how to solve. Ex.unweighted shortest path → weighted (set weights = 1)*

Examples Of Dynamic Programming

Ex. 1 : Given a string, find the longest substring which is palindrome.

Method 1 : Brute Force(Naive Approach)

Approach : *The simple approach is to check each substring whether the substring is a palindrome or not. To do this first, run three nested loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.*

```
# A Python3 solution for longest palindrome

# Function to pra subString str[low..high]
def printSubStr(str, low, high):

    for i in range(low, high + 1):
        print(str[i], end = "")

# This function prints the
# longest palindrome subString
# It also returns the length
# of the longest palindrome
def longestPalSubstr(str):

    # Get length of input String
    n = len(str)

    # All subStrings of length 1
    # are palindromes
    maxLength = 1
    start = 0

    # Nested loop to mark start
    # and end index
    for i in range(n):
        for j in range(i, n):
            flag = 1

            # Check palindrome
            for k in range(0, ((j - i) // 2) + 1):
                if (str[i + k] != str[j - k]):
                    flag = 0

            # Palindrome
            if (flag != 0 and (j - i + 1) > maxLength):
                start = i
                maxLength = j - i + 1

    print("Longest palindrome subString is: ", end = "")
    printSubStr(str, start, start + maxLength - 1)

    # Return length of LPS
    return maxLength

# Driver Code
if __name__ == '__main__':

    str = "forgeeksskeegfor"

    print("\nLength is: ", longestPalSubstr(str))
```

Output:

```
Longest palindrome subString is: geeksskeeg
Length is:  10
```

Complexity Analysis:

Time complexity: O(n^3).
Three nested loops are needed to find the longest palindromic substring in this approach, so the time complexity is O(n^3).
Auxiliary complexity: O(1).
As no extra space is needed.

Method 2 : *Dynamic Programming*

Approach : *The time complexity can be reduced by storing results of sub-problems.*
Following is a general recursive solution with all cases handled.

```
// Every single character is a palindrome of length 1
L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same
If (X[i] != X[j])  L(i, j) =  max{L(i + 1, j),L(i, j - 1)}

// If there are only 2 characters and both are same
Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last
// characters are same
Else L(i, j) =  L(i + 1, j - 1) + 2
```

Output:

```
The length of the LPS is 7
```

Time Complexity of the above implementation is O(n^2) which is much better than the worst-case time complexity of Naive Recursive implementation.

```python
# A Dynamic Programming based Python
# program for LPS problem Returns the length
#  of the longest palindromic subsequence in seq
def lps(str):
    n = len(str)

    # Create a table to store results of subproblems
    L = [[0 for x in range(n)] for x in range(n)]

    # Strings of length 1 are palindrome of length 1
    for i in range(n):
        L[i][i] = 1

    # Build the table. Note that the lower
    # diagonal values of table are
    # useless and not filled in the process.
    # The values are filled in a
    # manner similar to Matrix Chain
    # Multiplication DP solution (See
    # https://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-
multiplication/
    # cl is length of substring
    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if str[i] == str[j] and cl == 2:
                L[i][j] = 2
            elif str[i] == str[j]:
                L[i][j] = L[i+1][j-1] + 2
            else:
                L[i][j] = max(L[i][j-1], L[i+1][j]);

    return L[0][n-1]

# Driver program to test above functions
seq = "GEEKS FOR GEEKS"
n = len(seq)
print("The length of the LPS is " + str(lps(seq)))
```

Ex. 2 : Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

1) Insert
2) Remove
3) Replace
All of the above operations are of equal cost.

Example :
Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.
Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.
Input: str1 = "sunday", str2 = "saturday"
Output: 3
Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.
Replace 'n' with 'r', insert t, insert a
Steps for solving problem :
1) If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2)Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values :
i)Insert: Recur for m and n-1
ii)Remove: Recur for m-1 and n
iii)Replace: Recur for m-1 and n-1
Output
7
Time Complexity: O(m x n)
Auxiliary Space: O(m x n)

```
def minDis(s1, s2, n, m, dp) :

  # If any string is empty,
  # return the remaining characters of other string
  if(n == 0) :
    return m
  if(m == 0) :
    return n

  # To check if the recursive tree
  # for given n & m has already been executed
  if(dp[n][m] != -1)  :
    return dp[n][m];

  # If characters are equal, execute
  # recursive function for n-1, m-1
  if(s1[n - 1] == s2[m - 1]) :
   if(dp[n - 1][m - 1] == -1) :
     dp[n][m] = minDis(s1, s2, n - 1, m - 1, dp)
     return dp[n][m]
   else :
     dp[n][m] = dp[n - 1][m - 1]
     return dp[n][m]

  # If characters are nt equal, we need to
  # find the minimum cost out of all 3 operations.
  else :
   if(dp[n - 1][m] != -1) :
    m1 = dp[n - 1][m]
   else :
    m1 = minDis(s1, s2, n - 1, m, dp)

   if(dp[n][m - 1] != -1) :
    m2 = dp[n][m - 1]
   else :
    m2 = minDis(s1, s2, n, m - 1, dp)
   if(dp[n - 1][m - 1] != -1) :
    m3 = dp[n - 1][m - 1]
   else :
    m3 = minDis(s1, s2, n - 1, m - 1, dp)

   dp[n][m] = 1 + min(m1, min(m2, m3))
   return dp[n][m]

   # Driver code
str1 = "voldemort"
str2 = "dumbledore"

n = len(str1)
m = len(str2)
dp = [[-1 for i in range(m + 1)] for j in range(n + 1)]

print(minDis(str1, str2, n, m, dp))
```

Ex. 3 : Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.
Note: The opponent is as clever as the user.

Let us understand the problem with few examples:
1) 5, 3, 7, 10 : The user collects maximum value as 15(10 + 5)
2) 8, 15, 3, 7 : The user collects maximum value as 22(7 + 15)
Does choosing the best at each move gives an optimal solution? No.
In the second example, this is how the game can be finished:
1) …….User chooses 8.
…….Opponent chooses 15.
…….User chooses 7.
…….Opponent chooses 3.
Total value collected by user is 15(8 + 7)
2)…….User chooses 7.
…….Opponent chooses 8.
…….User chooses 15.
…….Opponent chooses 3.
Total value collected by user is 22(7 + 15)
So if the user follows the second game state, the maximum value can be collected although the first move is not the best.
.
Approach: As both the players are equally strong, both will try to reduce the possibility of winning of each other. Now let's see how the opponent can achieve this.
There are two choices:

1. The user chooses the 'ith' coin with value 'Vi': The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with minimum value.
i.e. The user can collect the value Vi + min(F(i+2, j), F(i+1, j-1)).
2. The user chooses the 'jth' coin with value 'Vj': The opponent either chooses 'ith' coin or '(j-1)th' coin. The opponent intends to choose the coin which leaves the user with minimum value, i.e. the user can collect the value Vj + min(F(i+1, j-1), F(i, j-2)).
.
Following is the recursive solution that is based on the above two choices. We take a maximum of two choices.

F(i, j) represents the maximum value the user
can collect from i'th coin to j'th coin.

```
F(i, j) = Max(Vi + min(F(i+2, j), F(i+1, j-1) ),
              Vj + min(F(i+1, j-1), F(i, j-2) ))
```
As user wants to maximise the number of coins.

Base Cases
```
    F(i, j) = Vi           If j == i
    F(i, j) = max(Vi, Vj)  If j == i + 1
```
Output:

```
22
4
42
```
Complexity Analysis:

Time Complexity: O(n2).
Use of a nested for loop brings the time complexity to n2.
Auxiliary Space: O(n2).
As a 2-D table is used for storing states.

```python
# Python3 program to find out maximum
# value from a given sequence of coins

# Returns optimal value possible that
# a player can collect from an array
# of coins of size n. Note than n
# must be even
def optimalStrategyOfGame(arr, n):

    # Create a table to store
    # solutions of subproblems
    table = [[0 for i in range(n)]
             for i in range(n)]

    # Fill table using above recursive
    # formula. Note that the table is
    # filled in diagonal fashion
    # (similar to http://goo.gl / PQqoS),
    # from diagonal elements to
    # table[0][n-1] which is the result.
    for gap in range(n):
        for j in range(gap, n):
            i = j - gap

            # Here x is value of F(i + 2, j),
            # y is F(i + 1, j-1) and z is
            # F(i, j-2) in above recursive
            # formula
            x = 0
            if((i + 2) <= j):
                x = table[i + 2][j]
            y = 0
            if((i + 1) <= (j - 1)):
                y = table[i + 1][j - 1]
            z = 0
            if(i <= (j - 2)):
                z = table[i][j - 2]
            table[i][j] = max(arr[i] + min(x, y),
                              arr[j] + min(y, z))
    return table[0][n - 1]

# Driver Code
arr1 = [ 8, 15, 3, 7 ]
n = len(arr1)
print(optimalStrategyOfGame(arr1, n))

arr2 = [ 2, 2, 2, 2 ]
n = len(arr2)
print(optimalStrategyOfGame(arr2, n))

arr3 = [ 20, 30, 2, 2, 2, 10]
n = len(arr3)
print(optimalStrategyOfGame(arr3, n))
```

Cryptographic Hashing

Hashing is a method of cryptography that converts any form of data into a unique string of text.

Amortization

Amortized analysis is a powerful technique for data structure analysis, involving the total runtime of a sequence of operations, which is often what we really care about.

Different techniques of amortized analysis :

Aggregate Method : *just add up the cost of all the operations and then divide by the number of operations.*
Amortized cost per operation = total cost of k operations / k
** Aggregate method is the simplest method. Because it's simple, it may not be able to analyze more complicated algorithms.*

Accounting Method : *This method allows an operation to store credit into a bank for future use, if its assigned amortized cost > its actual cost; it also allows an operation to pay for its extra actual cost using existing credit, if its assigned amortized cost < its actual cost.*

Charging Method : *The charging method allows operations to charge cost retroactively to past operations.*
Amortized cost of an operation = actual cost of this operation − total cost charged to past operations + total cost charged by future operations

Potential Method : *This method defines a potential function Φ that maps a data structure (DS) configuration to a value. This function Φ is equivalent to the total unused credits stored up by all past operations (the bank account balance). Now*
amortized cost of an operation = actual cost of this operation + ΔΦ

Examples of amortized analysis :
– table doubling
– binary counter
– 2-3 tree and 2-5 tree

Processor Architecture

Computer architecture has evolved:

- *Intel 8086 (1981): 5 MHz (used in first IBM PC)*
- *Intel 80486 (1989): 25 MHz (became i486 because of a court ruling that prohibits the trademarking of numbers)*
- *Pentium (1993): 66 MHz*
- *Pentium 4 (2000): 1.5 GHz (deep ≈ 30-stage pipeline)*
- *Pentium D (2005): 3.2 GHz (and then the clock speed stopped increasing)*
- *Quadcore Xeon (2008): 3 GHz (increasing number of cores on chip is key to performance scaling)*