Final Exam Notes

Time Complexity

Time complexity in algorithms is a measure that gives us an estimation of the time an algorithm takes to run as a function of the length of the input. It is expressed using Big O notation, which provides an upper bound on the growth rate of the runtime of an algorithm. Time complexity is important because it helps us predict the scalability of an algorithm and understand how it will perform as we work with larger datasets or more complex problems.

Big $\mathcal{O}$ Notation

Big $\mathcal{O}$ Notation is used to describe the upper bound of the algorithm's runtime complexity, indicating the worst-case scenario. Common types of runtimes are:

- $\mathcal{O}(1)$ **- Constant Time**: Time to complete is the same regardless of input size.
- $\mathcal{O}(\log(n))$ **- Logarithmic Time**: Time to complete increases logarithmically as input size increases.
- $\mathcal{O}(n)$ **- Linear Time**: Time to complete scales linearly with the input size.
- $\mathcal{O}(n\log(n))$ **- Linearithmic Time**: Time to complete is a combination of linear and logarithmic growth rates.
- $\mathcal{O}(n^2)$ **- Quadratic Time**: Time to complete scales with the square of the input size.
- $\mathcal{O}(2^n)$ **- Exponential Time**: Time to complete doubles with each additional input unit.

Big $\mathcal{O}$, Big Omega Ω, and Big Theta Θ represent different bounds of runtime complexity:

- **Big** $\mathcal{O}$: The upper bound, or worst-case complexity.
- **Big Omega** Ω: The lower bound, or best-case complexity.
- **Big Theta** Θ: An algorithm is Θ if it is both $\mathcal{O}$ and Ω, indicating a tight bound where the upper and lower bounds are the same.

Identifying Leading Terms

For a given algorithmic complexity, there is a recipe for identifying what the 'leading term' of an algorithm is. This helps us identify how fast an algorithm may grow as time goes on. The recipe for doing this is:

1. **Identify the Leading Term**: For large values of n the term with the highest exponent in n will have the most significant impact on the growth rate of the function. Constant factors are irrelevant in Big $\mathcal{O}$ notation.
2. **Simplify Logarithmic Expressions**: Convert all logarithms to the same base if possible, and remember that constants in front of logarithms (like 2 in $\log_2(n)$) do not change the complexity class. Use the change of base formula if needed.
3. **Compare Growth Rates**: Know the order of growth rates: $\log(n) < n < n\log(n) < n^k < a^n < n!$. This helps in quickly identifying which terms dominate as n grows large.
4. **Ignore Lower Order Terms and Coefficients**: When determining the Big $\mathcal{O}$ class, you can ignore any constants and any terms that grow more slowly than the leading term. For example, in $n^2 + 100n$, the $100n$ term is irrelevant for large n, and the function is $\mathcal{O}(n^2)$.
5. **Be Mindful of Exponentials**: Recognize that any exponential function a^n (where $a > 1$) will grow faster than any polynomial, and thus does not belong to $\mathcal{O}(n^k)$ for any constant k.
6. **Special Cases**: Be aware of functions that may look complex but simplify to a known growth rate, such as polynomial functions with non-integer exponents. Assess whether the polynomial growth dominates the logarithmic or constant factors.

Asymptotes

In the context of algorithms, the concept of asymptotes is related to the idea of asymptotic analysis, which is concerned with the behavior of algorithms as the size of the input grows very large. Here's a concise summary of how the concept relates to algorithms:

- **Asymptotic Behavior**: Asymptotic analysis looks at the limit of an algorithm's performance (time or space required) as the input size approaches infinity. It helps in understanding the efficiency of an algorithm in the worst case (usually).
- **Asymptotic Upper Bound (Big $\mathcal{O}$)**: This is like a 'ceiling' for the growth of an algorithm's running time. It means that the algorithm's running time will not exceed a certain boundary as the input size grows indefinitely.
- **Asymptotic Lower Bound (Big Ω)**: Analogous to a 'floor', it gives a guarantee that the algorithm's running time will be at least as high as the bound for sufficiently large inputs.
- **Tight Bound (Big Θ)**: If an algorithm has a Big Theta bound, it means that both the upper and lower bounds are the same asymptotically. The algorithm's running time grows at the same rate as the function in the Big Theta notation.
- **Asymptotic Equality**: When we say an algorithm has a time complexity of, say, $\mathcal{O}(n^2)$, we mean that the running time increases at most quadratically as n approaches infinity. We're not concerned with the exact match but the trend as n becomes very large.

Sorting Algorithms

Some of the main algorithms that were covered in this course are:

- **Bubble Sort**: Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order, leading to larger elements 'bubbling' to the top of the list with each iteration.
- **Insertion Sort**: Insertion sort is a comparison-based sorting algorithm that builds a final sorted array one element at a time by repeatedly taking the next element from the input data and inserting it into the correct position within the already sorted portion of the array.
- **Merge Sort**: Merge sort is a divide-and-conquer algorithm that divides the list into halves, recursively sorts each half, and then merges the sorted halves back together into a single sorted list.
- **Quick Sort**: Quick sort is an efficient sorting algorithm that uses a divide-and-conquer approach to select a 'pivot' element and partitions the other elements into two subarrays, those less than the pivot and those greater, before recursively sorting the subarrays.

The time complexities of these algorithms are

	Average Case	**Best Case**	**Worst Case**
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Merge Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$
Quick Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$

For each of these algorithms, they all have a worst case and best case scenario. The scenarios for these algorithms are:

- **Bubble Sort**: The worst-case runtime for bubble sort occurs when the original list is sorted in descending order, requiring the maximum number of swaps, while the best-case scenario is when the list is already sorted in ascending order, allowing the algorithm to complete with minimal comparisons and no swaps.
- **Insertion Sort**: The worst-case runtime for insertion sort occurs when the list is sorted in descending order, as each new element must be compared with all other sorted elements before being placed at the beginning, while the best-case scenario is when the list is already sorted in ascending order, allowing each new element to be placed without any further comparisons.
- **Merge Sort**: For merge sort, the worst-case and best-case runtimes are the same regardless of the initial order of the list, as the algorithm consistently divides the list and merges it in a systematic manner, resulting in a predictable and stable runtime.

- **Quick Sort**: The worst-case runtime for quick sort occurs when the list is already sorted in either ascending or descending order, causing the algorithm to degenerate into a linear scan at each recursive step if the pivot chosen is always the smallest or largest element, while the best-case scenario is when the pivot consistently divides the list into equal halves, optimizing the depth of recursive calls.

Master Method

The Master Method provides a method to analyze the time complexity of recursive algorithms, especially those following the divide and conquer approach.

Master Method Formula

The theorem applies to recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. $b > 1$.
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and combining the results of the subproblems.

The solution to the recurrence, $T(n)$, can be categorized into three cases based on the comparison between $f(n)$ and $n^{\log_b(a)}$ (the critical part of the non-recursive work). The constant ϵ is equated to be $\epsilon = \log_b(a)$, c is often referred to as the exponent of the leading term in $f(n)$.

1. **Case 1 (Divide or Conquer dominates)**
   - If $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$, **essentially** $\epsilon > c$.
     - Then $T(n) = \Theta(n^{\epsilon})$
2. **Case 2 (Balanced)**
   - If $f(n) = \Theta(n^{\log_b(a)} \cdot \log^k(n))$ for some constant $k \geq 0$, **essentially** $\epsilon = c$.
     - Then $T(n) = \Theta(n^{\epsilon} \log(n))$
3. **Case 3 (Work outside divide/conquer dominates)**
   - If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some constant $\epsilon > 0$, and if $af\left(\frac{a}{b}\right) \leq kf(n)$ for some constant $k < 1$ and large enough n, **essentially** $\epsilon < c$.
     - Then $T(n) = \Theta(f(n))$

Median Of Medians

The Median of Medians is a selection algorithm that serves as a trick to find a good pivot for sorting and selection algorithms, such as Quick Sort, or for solving the selection problem (finding the k-th smallest element) in linear time. The algorithm is particularly useful because it guarantees a pivot that is a 'good' approximation of the median, ensuring that the partition of the array is reasonably balanced, which helps avoid the worst-case scenario of $\mathcal{O}(n^2)$ time complexity in Quick Sort or other selection algorithms.

Median Of Medians Procedure

The procedure for performing the Median Of Medians is

1. **Grouping**: Divide the array into subarrays of n elements each, where n could be any small constant. The last group may have fewer than n elements if the size of the array is not a multiple of n.
2. **Find Medians**: Compute the median of each of these subarrays. If n is small, this can be done quickly through a brute-force approach.

3. **Recursive Median Selection**: Use the Median of Medians algorithm recursively to find the median of these n-element medians. This median will be used as the pivot.
4. **Partition Using the Pivot**: The chosen pivot divides the original array into parts such that one part contains elements less than the pivot, and the other part contains elements greater than the pivot.
5. **Recursive Application for Selection/Sorting**: Depending on whether you're sorting or selecting (e.g., finding the k-th smallest element), proceed with the algorithm recursively on the relevant partition(s).

The Median Of Medians follows a recursive formula of

$$T(n) = T(\alpha) + T(\beta) + \Theta(n)$$

where α is the part for finding true median of how many medians. For a median of m medians, we can calculate α and β with

$$\alpha = \frac{n}{m}$$
$$\beta = \left(1 - \frac{1}{2}\left(\frac{\lceil m/2 \rceil}{m}\right)\right) n.$$

Trees

Binary Search Tress (BST)

A Binary Search Tree (BST) is a node-based binary tree data structure with the following essential properties:

- Each node has at most two children, referred to as the left child and the right child.
- For any given node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node. This property is recursively true for all nodes in the tree.

These characteristics enable efficient performance of operations such as search, insertion, and deletion, with average and best-case time complexities of $\mathcal{O}(\log n)$, where n is the number of nodes in the tree.

Height of a Binary Search Tree

The height of a BST is measured as the number of edges on the longest path from the root node to a leaf node. It is a critical metric that influences the efficiency of various tree operations.

Height of a Node in a Binary Search Tree

The height of a node within a BST is determined by the number of edges on the longest path from that node to a leaf node in its subtree. The calculation follows the same recursive logic as the height of the entire tree, but starts from the specific node in question.

Red-Black Trees

Red-Black Trees are a type of self-balancing binary search tree, where each node contains an extra bit for denoting the color of the node, either red or black. This structure ensures the tree remains approximately balanced, leading to improved performance for search, insertion, and deletion operations. The properties of Red-Black Trees enforce constraints on node colors to maintain balance and ensure operational complexity remains logarithmic.

Properties Of Red-Black Trees

Red-Black Trees adhere to the following five essential properties:

1. **Node Color**: Every node is either red or black.
2. **Root Property**: The root node is always black.
3. **Red Node Property**: Red nodes must have black parent and black children nodes (i.e., no two red nodes can be adjacent).

4. **Black Height Property**: Every path from a node (including root) to any of its descendant NULL nodes must have the same number of black nodes. This consistent number is called the black height of the node.
5. **Path Property**: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

These properties ensure that the longest path from the root to a leaf is no more than twice as long as the shortest path, meaning the tree remains approximately balanced.

Runtime Complexity

The runtime complexities of Red-Black trees are the following:

Operation	Complexity
Delete	$\mathcal{O}(\log(n))$
Insert	$\mathcal{O}(\log(n))$
Search	$\mathcal{O}(\log(n))$

Hash Tables

Hash tables, are a type of data structure that implements an associative array, a structure that can map keys to values. A hashtable uses a hash function to compute an index into an array of slots, from which the desired value can be found. This approach enables efficient data retrieval, insertion, and deletion operations.

Key Components

- **Hash Function**: A function that computes an index (the hash) into an array of buckets or slots, from which the desired value can be found. The efficiency of a hash table depends significantly on the hash function it uses.
- **Buckets or Slots**: The array where the actual data is stored. Each slot can store one or more key-value pairs.
- **Collision Resolution**: Since a hash function may assign the same hash for two different keys (a collision), mechanisms such as chaining (linked lists) or open addressing (probing) are used to resolve collisions.

Operations

1. **Insertion**: Add a new key-value pair to the table.
2. **Deletion**: Remove a key-value pair from the table.
3. **Lookup**: Retrieve the value associated with a given key.

The average-case time complexity for these operations is $\mathcal{O}(1)$, assuming a good hash function and low load factor, making hashtables one of the most efficient data structure for these types of operations.

Runtime Complexity

The runtime complexities for the operations of hash tables are:

Operation	Average Case	Best Case	Worst Case
Deletion	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Lookup	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

For a hash table that has n buckets (slots) in the hash table, the **maximum number of keys** that can be mapped to the buckets in the hash table **without a collision** is n.

Heaps: Min Heaps and Max Heaps

Heaps are a specialized tree-based data structure that satisfy the heap property. In a Min Heap, for any given node, the value of the node is less than or equal to the values of its children. Conversely, in a Max Heap, for any given node, the value of the node is greater than or equal to the values of its children. Heaps are commonly used to implement priority queues and for efficient sorting (Heap Sort).

Heap Representation

Heaps are often represented as arrays for efficiency, with the relationships between parent nodes and children nodes implicitly defined by their indices in the array.

Node Relationships in an Array Representation

Given a node at index i in the array:

- The index of the left child is $2i + 1$.
- The index of the right child is $2i + 2$.
- The index of the parent node is $\lfloor (i - 1)/2 \rfloor$, for any node except the root.

Operations

Common operations of heaps are:

- **Deletion of Root**: Remove the root element (which is the maximum in a max-heap or minimum in a min-heap). After removing the root, the last element in the heap is temporarily moved to the root position and then 'bubbled down' (or sifted down) to restore the heap property.
- **Increase / Decrease Key**: This operation modifies the value of an element in the heap. Depending on whether it's an increase in a max-heap or a decrease in a min-heap, the element might need to be bubbled up to maintain the heap structure. Conversely, for a decrease in a max-heap or an increase in a min-heap, the element might need to be bubbled down.
- **Insertion**: Add a new element to the heap while maintaining the heap property. The element is initially inserted at the end of the heap (the last position of the array), and then it is 'bubbled up' (or sifted up) to its correct position in the heap by comparing it with its parent and swapping if necessary.
- **Find Max / Min**: Retrieve the maximum element in a max-heap or the minimum element in a min-heap, which is always at the root of the heap.
- **Heapify**: Transform an arbitrary array into a heap. This process involves rearranging the elements of the array so that they satisfy the heap property.

Runtime Complexity

The runtime complexities of heaps are:

Operation	Average Case	Best Case	Worst Case
Deletion of Root	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Increase / Decrease	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Insertion	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Find Max / Min	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Heapify	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Graphs

Graphs are a fundamental data structure in computer science and mathematics, extensively used to model relationships and pathways. They are particularly useful for representing networks such as social connections, telecommunications, computer networks, and road maps.

Basic Concepts

The core tenants of graphs are the following:

- **Edges (Links)**: These are the connections between vertices, which can be directed or undirected, representing the relationships or pathways between the entities.
- **Vertices (Nodes)**: These are the fundamental units or points in a graph, representing entities such as cities, computers, people, etc.

Properties of Graphs

Graphs all have common properties, here are some common properties found in graphs:

- **Acyclic Graph**: A graph without any cycles.
- **Adjacency**: Two vertices are adjacent if they are connected by an edge.
- **Connected Graph**: There is a path between every pair of vertices in the graph.
- **Cycle**: A path that starts and ends at the same vertex without repeating any edge.
- **Path**: A sequence of vertices where each adjacent pair is connected by an edge.

Types of Graphs

In graph theory, there are a multitude of different types of graphs. Here are the main types:

- **Directed Acyclic Graphs (DAGs)**: Directed acyclic graphs (DAGs) are directed graphs where no cycle is present.
- **Directed Graphs (Digraphs)**: Here, edges have a direction. If an edge is directed from A to B, you can only traverse from A to B, not the other way around unless there's a corresponding inverse edge.
- **Undirected Graphs**: In these graphs, edges have no direction. If there is an edge between vertex A and vertex B, you can traverse from A to B and from B to A without any restriction. This type is often used to represent bi-directional relationships.
- **Unweighted Graphs**: Edges do not carry any weight. The connections simply represent a binary relationship, either connected or not.
- **Weighted Graphs**: These graphs have edges that carry a weight or cost, useful for representing routes with distances, costs, or capacities, etc.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a traversal algorithm for graphs that explores vertices in layers, ensuring that all nodes at the current depth (distance from the starting point) are explored before moving on to nodes at the next depth level. This method is particularly useful for finding the shortest path on unweighted graphs, where all edges have the same weight, or for traversing a graph in a way that naturally aligns with level order (such as in trees).

BFS Procedure

Here is the general process for performing a Breadth-First Search (BFS) on a graph:

1. **Start by picking a source node**: This node is where the BFS will begin.
2. **Initialize a queue**: Add the starting node to the queue. This queue will help manage nodes as you explore them.
3. **Mark the source node as visited**: You can use a list or a set to keep track of which nodes have been visited. Initially, this list will only contain the starting node.
4. **Process the queue**:
   - **Dequeue a node from the front of the queue**: This is the current node you are exploring.
   - **Check each of its adjacent nodes**: For each adjacent node, determine if it has been visited.
     - If it hasn't been visited, mark it as visited and enqueue it at the back of the queue. This step ensures that nodes are visited level by level.
   - **Repeat this process until the queue is empty**.
5. **Continue until all possible nodes are visited or the queue is empty**: BFS ensures that once the queue is empty, all nodes reachable from the starting node have been explored.

Runtime Complexity

The runtime complexity for BFS is $\mathcal{O}(|V| + |E|)$ where V is the number of vertices and E is the number of edges in the graph.

Depth First Search (DFS)

Depth-First Search (DFS) is a fundamental algorithm used to traverse or search through a graph or tree structure. It explores as far as possible along each branch before backtracking, making it especially useful for problems involving paths, cycles, connectivity, and discovering the structure of a maze or puzzle.

DFS Procedure

Here is the general process for performing a Depth-First Search (DFS) on a graph:

1. **Choose a starting node**: Begin from a specified node which acts as the root for a DFS traversal.
2. **Initialize a stack**: If implementing iteratively, use a stack to manage nodes to explore. For recursive implementations, the call stack manages this inherently.
3. **Mark the starting node as visited**: Maintain a set or list to keep track of visited nodes to prevent re-visiting and looping.
4. **Explore the graph**:
   - **Add the starting node to the stack**: Place the initial node into the stack.
   - **Loop until the stack is empty**:
     - **Pop a node from the stack**: This is your current node.
     - **Visit this node**: You can process or print the node as needed.
     - **Check each adjacent node**: For each neighbor of this node:
       * If the neighbor has not been visited, mark it as visited and push it onto the stack. This step ensures that the traversal dives deeper into the graph following one path until it can't go further.
     - **Backtrack when necessary**: When a node has no unvisited neighbors, the stack's nature causes the algorithm to naturally backtrack, moving back to explore other branches.
5. **Repeat until the stack is empty**: The process continues until every reachable node from the starting point has been explored.

Tree, Back, Forward, and Cross Edges (in the context of DFS)

Depth-First Searches have different types of edges in their traversals. Here are the four types of edges that are present in a DFS:

- **Tree Edges**: In a DFS forest, tree edges are those that are part of the original graph and connect vertices to their descendants in the DFS tree.
- **Back Edges**: These are edges that point from a node to one of its ancestors in the DFS tree. Back edges are critical for detecting cycles in directed graphs.
- **Forward Edges**: These edges connect a node to a descendant in the DFS tree, but they are not tree edges. They essentially skip over parts of the tree.
- **Cross Edges**: These are edges that connect nodes across branches of the DFS tree, neither to ancestors nor to descendants.

When performing a DFS on a graph, some edges are not possible in certain graphs:

- In the context of **DAGs**, **Back Edges** are not possible to be present in the traversal.
- In the context of **Undirected Graphs**, **Cross Edges** are not possible to be present in the traversal.

Strongly Connected Components (SCCs)

Strongly Connected Components (SCCs) pertain to the field of graph theory and are particularly applicable to directed graphs. An SCC is defined as a maximal subgraph of a directed graph in which every vertex is reachable from every other vertex within the same component.

Key Concepts

SCCs consist of some core concepts, here are the two main concepts in regards to SCCs:

- **Directed Graphs**: SCCs are a concept specific to directed graphs where the direction of edges affects the connectivity.
- **Reachability**: In a strongly connected component, there must be a directed path from every node to every other node within the same component.

Properties Of SCCs

The core tenants of SCCs are:

- **Maximal**: No additional vertices can be included in an SCC without breaking the property of mutual reachability.
- **Meta-graph**: If each SCC is contracted to a single node, the resulting graph (often called the component graph or meta-graph) is a Directed Acyclic Graph (DAG).
- **Partition**: The set of all SCCs in a graph partitions the vertices of the graph; every vertex belongs to exactly one SCC.

A Maximal Strongly Connected Component (MSCC) is a subset of a SCC where the MSCC contains the maximum number of vertices (nodes) in it possible. In regards to graphs:

- **The maximum number of MSCCs** in a graph is **the number of nodes** in the graph. This is because each node that is not part of a MSCC already has the potential of being its own MSCC.
- **The minimum number of MSCCs** in a graph is **one**. This is because the entire graph can be a SCC and thus its own MSCC.

Shortest Path Algorithms

Shortest path algorithms are crucial in graph theory and are widely used in various applications, from routing and network design to logistics and urban planning. These algorithms aim to find the shortest path between two nodes in a graph, which can either be unweighted or weighted.

Dijkstra's Algorithm

Dijkstra's algorithm is an example of a shortest path algorithm. The main ideas behind Dijkstra's algorithm are:

- **Main Idea**: Uses a priority queue to greedily select the next vertex with the minimal distance; updates distances for its adjacent vertices.
- **Graph Type**: Works on graphs with non-negative edge weights.

The runtime complexities for Dijkstra's algorithm are:

Simple Array	Heap
$\mathcal{O}(V^2)$	$\mathcal{O}((V+E)\log(V))$

Bellman-Ford Algorithm

The Bellman-Ford algorithm is another example of a shortest path algorithm. The main ideas behind the Bellman-Ford algorithm are:

- **Main Idea**: Relaxes edges repeatedly to update the shortest paths from a source vertex to all other vertices in the graph, allowing for up to $V-1$ relaxations where V is the number of vertices.
- **Graph Type**: Can handle graphs with negative edge weights, but not negative cycles.

The runtime complexity for the Bellman-Ford algorithm is $\mathcal{O}(V \cdot E)$.

Floyd-Warshall Algorithm

Similar to both Dijkstra's and the Bellman-Ford algorithms, the Floyd-Warshall algorithm is another shortest path algorithm. The tenants of the Floyd-Warshall algorithm are:

- **Main Idea**: A dynamic programming approach that calculates the shortest paths between all pairs of vertices.
- **Graph Type**: Handles both negative and positive weights and can detect negative cycles.

The runtime complexity for the Floyd-Warshall Algorithm is $\mathcal{O}(V^3)$.

Minimum Spanning Tree (MST)

A Minimum Spanning Tree (MST) of a weighted graph is a subset of the edges that forms a tree connecting all vertices without any cycles and with the minimum possible total edge weight.

Key Properties Of MSTs

MSTs adhere to the following properties:

- **Connectivity**: Connects all vertices in the graph without cycles.
- **Minimum Weight**: The total weight of all edges in the tree is as small as possible.
- **Uniqueness**: The MST is unique if all the edge weights are distinct; otherwise, there may be multiple MSTs with the same minimum weight.

For a **graph of** n **vertices**, there are $n - 1$ **edges** in the MST.

Kruskal's Algorithm

One of the popular algorithms that are used for finding a MST is Kruskal's algorithm. The concepts of this algorithm are:

- **Approach**: A greedy algorithm that sorts all the edges by weight and adds the smallest edge to the growing forest, skipping edges that would form a cycle.
- **Graph Type**: Works well with graphs where edges are scattered broadly across the graph.

The time complexity of Kruskal's algorithm is $\mathcal{O}(V \log(V))$ or $\mathcal{O}(E \log(E))$.

Prim's Algorithm

Another popular algorithm that is used for finding the MST is Prim's algorithm. The main ideas of this algorithm are:

- **Approach**: Starts from a single vertex and grows the MST by repeatedly adding the cheapest edge that connects a vertex in the MST to a vertex outside the MST.
- **Graph Type**: Highly effective for dense graphs.

Dynamic Programming

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems, solving each of these subproblems just once, and storing their solutions—using these pre-computed solutions to construct the solution to the original problem. It is especially suited for problems exhibiting the properties of overlapping subproblems and optimal substructure.

Key Properties

The core tenants of Dynamic Programming are:

- **Optimal Substructure**: An optimal solution to the problem contains optimal solutions to the subproblems.
- **Overlapping Subproblems**: The problem can be broken down into subproblems which are reused several times.

When these properties are present, Dynamic Programming provides a powerful tool, reducing time complexity by trading computation for storage.

Dynamic Programming Approaches

In Dynamic Programming, there are two main approaches to solving the problem:

- **Bottom-Up Approach (Tabulation)**
  - This approach involves filling up a DP table based on the smallest subproblems, building up the solution to the overall problem.
  - Starts by solving the smallest subproblems first, using their solutions to iteratively solve more complex subproblems until the overall problem is solved.
  - Often more space-efficient than memoization and avoids the overhead of recursion.
  - Example Use: Solving the Knapsack problem, where the goal is to maximize the total value of items that can be carried, given a weight capacity.
- **Top-Down Approach (Memoization)**
  - This approach employs a method typically associated with recursion. It solves the main problem by recursively breaking it down into smaller and simpler subproblems.
  - These subproblems are solved once and their results are stored in a table (often an array or a hash table), so the same subproblem isn't solved more than once.
  - Example Use: Computing Fibonacci numbers where each number is the sum of the two preceding ones.

A common theme in Dynamic Programming solutions is when initializing quantities, they are either set to **zero** or **infinity**.

Non-deterministic Polynomial (NP) Problems

In computational complexity theory, the class NP (Non-deterministic Polynomial time) represents a set of problems for which any given solution can be verified quickly (in polynomial time) by a deterministic Turing machine. This class is significant because it includes many of the most common and challenging problems in computer science.

Definition Of NP, NP-Complete, And NP-Hard

We define NP, NP-Complete, and NP-Hard as:

- **NP (Non-deterministic Polynomial Time)**
  - Problems for which a solution, once given, can be verified in polynomial time by a deterministic Turing machine.
  - Example: The Hamiltonian Cycle problem, where given a set of vertices and edges, verifying if there is a cycle that visits each vertex exactly once can be done quickly.
- **NP-Complete**
  - A subset of NP that are at least as hard as any other problem in NP.
  - Any problem in NP can be reduced to any NP-complete problem in polynomial time.
  - If any NP-complete problem can be solved in polynomial time, then every problem in NP can also be solved in polynomial time, effectively proving P=NP.
  - These problems are used as benchmarks for the complexity of other problems in NP.
  - Example: The Traveling Salesman Problem (TSP), where determining the shortest possible route that visits each city once and returns to the origin city is NP-Complete.
- **NP-Hard**
  - Problems that are at least as hard as the hardest problems in NP.
  - An NP-Hard problem does not necessarily have to be in NP (i.e., it might not be possible to verify a solution in polynomial time).
  - These problems are not just limited to decision problems (which have a yes/no answer) but can include optimization and search problems.
  - If any NP-hard problem can be solved in polynomial time, then all NP problems can also be solved in polynomial time, but the reverse might not be true.
  - Example: The Halting Problem, which is about determining whether a given program will finish running or continue to run forever. It is NP-Hard but not necessarily in NP since verifying a 'no' answer (it does not halt) cannot be done in finite time.

Showing That A Problem Is NP

Below is a cookie cutter process for showing that a problem is NP:

1. Begin by clearly defining the problem, including specifying what constitutes an 'instance' of the problem and what constitutes a 'solution' to that instance. Ensure that the problem is decision-based (typically has yes/no answers), as this is a common form of problems in the NP class.
2. Describe the solution verification process
   - **Identify The Solution Certificate**: For many problems, a 'certificate' or 'witness' can be presented along with the instance. This certificate is a potential solution to the problem instance that needs to be verified.
3. Construct an algorithm that takes an instance of the problem and the accompanying certificate as input and checks whether the certificate actually solves the problem for that instance.
   - **Input**: The problem instance and the solution certificate.
   - **Output**: A boolean value (true or false) indicating whether the certificate correctly solves the instance.
4. Analyze the verification algorithm's Complexity
   - **Polynomial Time Verification**: Demonstrate that your verification algorithm runs in polynomial time relative to the size of the input. You will need to argue that the number of steps required to verify the solution is a polynomial function of the input size.
5. Show that this verification process applies to any instance of the problem and any possible solution certificate, not just specific cases. This involves arguing that no matter what the specific details of the instance or certificate are, the verification algorithm will correctly and efficiently determine if the certificate is a valid solution.
6. Conclude that since the problem has a verification algorithm that runs in polynomial time for any given solution certificate, the problem is in NP.

Showing That A Problem Is NP Complete

Below is a cookie cutter process for showing that a problem is NP Complete:

1. First, confirm that the problem is in NP, which means that for any given solution (certificate), the correctness of the solution can be verified in polynomial time. This step was detailed in the previous answer, where you develop and analyze a polynomial-time verification algorithm for potential solutions.
2. Choose a problem that has already been proven to be NP-complete. This is crucial because the NP-completeness proof typically involves a reduction from a known NP-complete problem. Common choices include:
   - **3-SAT**: A satisfiability problem where each clause has exactly three literals.
   - **Vertex Cover**: Finding a minimum set of vertices that cover all edges in a graph.
   - **Hamiltonian Cycle**: Determining whether a graph contains a cycle that visits each vertex exactly once.
3. Show that this known NP-complete problem can be transformed or reduced to the problem you are trying to prove is NP-complete. The reduction must meet the following criteria:
   - **Polynomial Time**: The transformation process itself must run in polynomial time, which means the time required to convert instances of the known NP-complete problem into instances of your problem should be bounded by a polynomial function of the size of the input.
   - **Preservation of Yes/No Answers**: If the instance of the known NP-complete problem has a solution (a 'yes' instance), then the transformed instance should also have a solution, and vice versa
4. Detail the steps of the reduction, illustrating how any instance of the known NP-complete problem can be systematically converted into an instance of your problem. This often involves constructing

instances of your problem that mimic the structure or constraints of the known problem while ensuring the essential properties are preserved.

5. Demonstrate that your reduction correctly transforms 'yes' instances to 'yes' instances and 'no' instances to 'no' instances. This usually requires:
   - **Forward Direction**: If the original problem instance is a 'yes' instance, you need to show that the constructed instance of your problem also admits a 'yes' answer.
   - **Reverse Direction**: Show that if the constructed problem instance has a 'yes' answer, then the original problem instance must also be a 'yes' instance.
6. Conclude that because every instance of the known NP-complete problem can be polynomially reduced to your problem, your problem must be at least as hard as the known NP-complete problem. Therefore, your problem is also NP-complete.

Showing That A Problem Is NP Hard

Below is a cookie cutter process for showing that a problem is NP Hard:

1. To prove a problem is NP-Hard, you start with an NP-complete problem. NP-complete problems are known to be among the hardest problems in NP because they can be used to simulate any other NP problem in polynomial time. Common examples include:
   - **3-SAT**: Where you decide if there exists an assignment to variables that satisfies all clauses, and each clause has exactly three literals.
   - **Traveling Salesman Problem**: Where you decide if there's a tour visiting each city exactly once with a total travel cost not exceeding a given limit.
   - **Hamiltonian Cycle**: Where you decide if there is a cycle in a graph that visits each vertex exactly once.
2. The critical step in proving a problem is NP-Hard is to provide a polynomial-time reduction from the chosen NP-complete problem to the problem you're analyzing. The reduction must transform instances of the NP-complete problem into instances of your target problem such that:
   - **Polynomial Time**: The transformation must be achievable in polynomial time relative to the size of the input.
   - **Solution Preservation**: If the original problem's instance is a 'yes' instance, then the transformed instance must also be a 'yes' instance, and vice versa for 'no' instances.
3. Clearly articulate how you can systematically convert any instance of the chosen NP-complete problem into an instance of your problem. This step is crucial as it forms the backbone of your proof:
   - Outline the steps of the transformation.
   - Explain how the properties and solutions of the NP-complete problem are preserved in the new instances of your problem.
4. This step involves proving that your reduction maintains the correctness in terms of 'yes' and 'no' answers:
   - **Forward Direction**: Demonstrate that if the original instance (from the NP-complete problem) is solvable (i.e., it's a 'yes'), then the transformed instance of your problem is also solvable.
   - **Reverse Direction**: Optionally, for some NP-Hard proofs, you may need to show that solving your problem also helps in solving the original NP-complete problem, though this is more critical when establishing equivalences for NP-completeness.
5. Conclude that since any problem in NP can be reduced to your problem, your problem is at least as hard as the hardest problems in NP, thus establishing it as NP-Hard.