College of Engineering & Applied Sciences

CSPB 3104

Algorithms

Exam Notes

Taylor Larrechea

2024

Exam 1 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.

Master's Method

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

Form of Recurrence

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 Three Cases

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 $T(n) = \Theta(f(n))$

Core Tenants Of Master's Method

- **Applicability**
  - The Master Method is particularly useful for analyzing the time complexity of algorithms that break problems down into smaller subproblems, each of a fixed proportion of the size of the original.
  - Common examples include merge sort, quicksort, and binary search algorithms.
- **Limitations**
  - It does not apply to all types of recurrences.
  - It cannot be used when the recursive subproblems are of unequal size.

Recursion Trees

Recursion trees provide a visual and intuitive method for solving recurrence relations in recursive algorithms, especially those in divide and conquer algorithms.

Concept

- A recursion tree is a tree where each node represents the cost of a certain part of the algorithm.
- The root represents the initial problem; children of a node represent subproblems that the algorithm divides the problem into.

Building a Recursion Tree

The recipe for building a recursion tree can be summarized below:

1. **Root Node**: Represents the initial problem size, n.
2. **Child Nodes**: Each level of the tree corresponds to a recursive call. The children of a node represent the subproblems into which the problem is divided.
3. **Cost at Each Node**: Write the cost of the work done at each level (excluding the recursive calls) on the corresponding node.
4. **Depth of the Tree**: Corresponds to the number of recursive calls before reaching the base case.

Analyzing the Tree

We can analyze a recursion tree with the following steps:

1. **Calculate Cost at Each Level**: Sum the costs of all nodes at each level of the tree.
2. **Total Cost**: The total cost of the algorithm is the sum of the costs at each level of the tree.

Exam 2 Notes

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 and Their Complexities

1. **Insertion**: Inserting a new element into a heap involves adding the element to the end of the array and then adjusting its position to maintain the heap property. This adjustment, or "heapifying up," has a time complexity of $\mathcal{O}(\log n)$.
2. **Deletion of Root**: Removing the root element (the minimum element in a Min Heap or the maximum element in a Max Heap) involves moving the last element in the array to the root position and then "heapifying down" to maintain the heap property. This operation also has a time complexity of $\mathcal{O}(\log n)$.
3. **Find Min/Max**: In a Min Heap or Max Heap, finding the minimum or maximum value, respectively, is a constant-time operation, $\mathcal{O}(1)$, as this value is always at the root of the heap.

Quick Sort

Quick Sort is a highly efficient sorting algorithm that utilizes the divide-and-conquer strategy. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process is repeated until the whole array is sorted.

Steps

1. **Pivot Selection**: The first step is to select a pivot element. There are multiple strategies for this, such as choosing the first element, the last element, the middle element, or even a random element from the array.
2. **Partitioning**: Rearrange the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this step, the pivot is in its final position. This is called the partition operation.
3. **Recursion**: Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with larger values.

Quick Sort

An example of the Quick Sort Algorithm can be seen in Python below

```python
def quick_sort(arr):
if len(arr) <= 1:
    return arr
else:
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

```
10
```

The above implementation is selecting the pivot element as the middle element of the given array.

Runtime Analysis

The runtime analysis of Quick Sort is as follows:

- **Best and Average Case**: The average and best-case performance of quick sort is $\mathcal{O}(n \log(n))$, where n is the number of elements in the array. This occurs when the pivot divides the array into two nearly equal halves, leading to a balanced tree.

- **Worst Case**: The worst-case scenario occurs when the pivot selection results in one partition being significantly smaller than the other, e.g., when the smallest or largest element is always chosen as the pivot. This leads to an unbalanced partition and gives a runtime of $\mathcal{O}(n^2)$. However, this can be mitigated by choosing a good pivot.

Quick Select

Quick Select is an efficient selection algorithm to find the k-th smallest element in an unordered list. It is closely related to the Quick Sort sorting algorithm, utilizing a similar partitioning approach. The key difference is that Quick Select only recurses into the part of the array that contains the k-th smallest element, reducing the average time complexity.

Steps

1. **Choose Pivot**: Select a pivot element from the array. The choice of pivot can be random or based on the Median of Medians strategy to ensure good average performance and avoid worst-case scenarios.

2. **Partition**: Rearrange the array such that all elements less than the pivot come before the pivot, while all elements greater than the pivot come after it. The pivot is then in its final position, with a certain number of elements preceding it.

3. **Target Check**: If the pivot's position is k, the pivot is the k-th smallest element, and the algorithm terminates. If the pivot's position is greater than k, repeat the algorithm on the sub-array of elements before the pivot. If the pivot's position is less than k, repeat on the sub-array of elements after the pivot, adjusting k accordingly.

Runtime Analysis

The average time complexity of Quick Select is $\mathcal{O}(n)$, making it highly efficient for selection problems. This efficiency stems from the fact that it only needs to process one partition of the array, unlike Quick Sort, which must sort both partitions.

- **Best and Average Case**: In the average and best-case scenarios, the partitioning divides the array into parts that diminish in size exponentially, leading to an overall linear time complexity.

- **Worst Case**: Without the Median of Medians strategy, the worst-case time complexity can degrade to $\mathcal{O}(n^2)$, particularly if the smallest or largest element is consistently chosen as the pivot. However, this is mitigated by intelligent pivot selection.

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.

Steps

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).

Runtime Analysis

The choice of n (the size of the groups) affects the constants in the time complexity but not the overall linear time complexity for the selection. The analysis hinges on two key observations:

- At least half of the medians are greater than (or equal to) the median-of-medians, and at least half are less than (or equal to) it.

- Because each median is greater (or smaller) than at least half the elements in its group, the pivot (median-of-medians) effectively guarantees that at least 1/4 of the elements are less than it and at least 1/4 are greater, ensuring a balanced split.

This balancing ensures that the algorithm makes significant progress in reducing the problem size at each step, leading to a linear time complexity, $\mathcal{O}(n)$, for the selection problem.

Binary Search Trees (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.

Operations Complexity

Due to the self-balancing nature of Red-Black Trees, operations such as insertion, deletion, and lookup can be performed in $\mathcal{O}(\log n)$ time complexity, where n is the number of nodes in the tree.

Hash Tables

Hashtables, also known as 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.

Universal Hash Functions

Universal hash functions are designed to minimize the probability of collisions between keys being hashed into the same slot or bucket. A family of hash functions $\mathcal{H}$ is considered universal if, for any two distinct keys k and l, the probability of a collision is at most $1/m$, where m is the number of slots in the hashtable.

Properties

- **Reduced Collisions**: By randomly choosing a hash function from a universal family at the beginning of execution, universal hash functions help distribute keys more uniformly across the buckets, reducing the chance of collisions.
- **Performance**: Universal hash functions ensure that the average-case time complexity of hashtable operations remains constant, $\mathcal{O}(1)$.

Runtime Complexity

The efficiency of hash table operations—namely insertion, deletion, and lookup—is a critical aspect of their performance. These operations ideally have a time complexity of $\mathcal{O}(1)$ on average. However, this efficiency can be affected by several factors, including the choice of hash function, the method of collision resolution, and the load factor of the hash table. Here we explore these factors in detail.

Factors Affecting Runtime Complexity

1. **Hash Function**: The hash function's role is to distribute keys uniformly across the buckets. A poor hash function can lead to clustering, where many keys hash to the same index, increasing the likelihood of collisions and thereby the time complexity of operations.
2. **Collision Resolution Strategy**: How collisions are handled significantly impacts performance. Two primary methods are:
    - **Chaining**: Each bucket at a given index stores a linked list of all elements that hash to the same index. While insertion remains $\mathcal{O}(1)$, deletion and lookup operations can degrade to $\mathcal{O}(n)$ in the worst-case scenario when all elements collide at the same index.
    - **Open Addressing**: All elements are stored within the array itself. If a collision occurs, the hash table probes for the next available slot according to a probing sequence. The worst-case time complexity can also degrade to $\mathcal{O}(n)$, particularly in a highly congested table.
3. **Load Factor** (α): Defined as $\alpha = \frac{n}{m}$, where n is the number of entries and m is the number of buckets. A higher load factor indicates a more filled table, increasing the likelihood of collisions and the costs associated with collision resolution, thus affecting the average-case complexity.

Average-case Time Complexity

Assuming a well-designed hash function and a low to moderate load factor, the average-case time complexity for insertion, deletion, and lookup operations in a hash table is $\mathcal{O}(1)$. This assumes that the collision resolution process is efficient and that the hash function distributes keys uniformly.

Worst-case Time Complexity

In the worst-case scenario, particularly with a poor hash function or high load factor, all keys could hash to the same index, leading to a situation where each operation takes $\mathcal{O}(n)$ time. This scenario is typically mitigated by ensuring a good hash function and keeping the load factor manageable through techniques like dynamic resizing.

Mitigation Strategies

- **Dynamic Resizing**: To maintain a low load factor, hash tables often increase their size and re-hash all entries when a certain load threshold is exceeded.
- **Choosing an Efficient Collision Resolution Method**: Depending on the use case, selecting between chaining and open addressing can optimize performance.
- **Using Universal Hash Functions**: These functions reduce the probability of collisions and help maintain the $\mathcal{O}(1)$ average-case time complexity.

Exam 3 Notes

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 - ideally, using a bottom-up approach. The key idea behind DP is to avoid recalculating answers to the subproblems that are solved multiple times by caching these results. DP can be applied to a wide range of problems, from optimization problems to counting and decision problems.

Key Concepts

1. **Overlapping Subproblems**: DP is used when the problem can be divided into subproblems which are reused several times.

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

3. **Memoization**: This is a top-down approach where you start solving the problem by breaking it down. If the problem has been solved, you would simply retrieve the stored answer. If not, solve it and store the answer.

4. **Tabulation**: A bottom-up approach where you solve all possible small problems and then combine them to build solutions for bigger problems.

Runtime and Spatial Complexity

The time and space complexity of DP algorithms vary greatly depending on the problem and the approach (memoization vs. tabulation). Generally, the complexity depends on the number of distinct states (subproblems) that need to be solved and stored.

- **Memoization**: The time complexity is often $\mathcal{O}(N *$ complexity of recursion), where N is the number of subproblems uniquely solved. The space complexity includes the storage for memoization and the call stack for recursion.

- **Tabulation**: The space complexity is usually similar to memoization, focusing on storing solutions of subproblems. However, it avoids the additional space required for the recursion call stack, making it more space-efficient for some problems.

Common Applications and Examples

1. **Fibonacci Series**: A classic example where DP can drastically reduce the time complexity from exponential in naive recursion to linear by storing the results of the Fibonacci numbers already calculated.

2. **Knapsack Problem**: Used to determine the maximum value that can be put in a knapsack of a given capacity by using a tabulated DP approach.

3. **Shortest Paths in Graphs (e.g., Floyd-Warshall, Bellman-Ford algorithms)**: DP is used to find shortest paths in a weighted graph with positive or negative edge weights but with no negative cycles.

4. **Coin Change Problem**: Determines the minimum number of coins that make a given value, using a bottom-up DP approach to build up solutions to all values up to the target.

Dynamic Programming is a powerful technique that requires identifying the problem's structure to effectively apply it. By understanding the concept of overlapping subproblems and optimal substructure, you can utilize DP to solve a variety of complex problems more efficiently.

Memoization

Memoization is a technique used in computing to speed up the execution of programs by storing the results of expensive function calls and reusing them when the same inputs occur again. It's a critical concept within Dynamic Programming (DP), enabling it to efficiently solve problems with overlapping subproblems. Here, we delve into memoization, focusing on its mechanism, complexities, and applications.

Mechanism

1. **Storage**: When a function is called, its result is stored in a data structure (e.g., an array or a hash table) with its parameters as the key.

2. **Lookup**: Upon subsequent calls with the same parameters, the function first checks the data structure. If the result is present, it's returned immediately, avoiding the recomputation.

3. **Recursive Calls**: Memoization is commonly applied in recursive algorithms where the same computations are repeated multiple times.

Runtime Complexity

The runtime complexity of a memoized algorithm depends on the number of unique states or subproblems to solve. If a problem has N unique states and the computation for each state is $\mathcal{O}(1)$ (excluding recursive calls), then the total runtime complexity would be $\mathcal{O}(N)$. This significantly reduces the time from what could be exponential without memoization in problems with overlapping subproblems.

Spatial Complexity

The space complexity is primarily determined by the number of unique function calls that need to be stored. This can vary from linear to potentially very high, depending on the problem's scope and the dimensions of the memoization table. The auxiliary space for the call stack should also be considered, especially for recursive solutions.

Applications and Examples

1. **Fibonacci Sequence**: Instead of computing Fibonacci numbers recursively in $\mathcal{O}(2^N)$ time, memoization allows solving it in $\mathcal{O}(N)$ by storing previously computed Fibonacci numbers.

2. **Longest Common Subsequence (LCS)**: Memoization can reduce the time complexity from exponential to $\mathcal{O}(M * N)$, where M and N are the lengths of the two sequences by caching the results of LCS lengths for different pairs of prefixes.

3. **Matrix Chain Multiplication**: Determines the most efficient way to multiply a series of matrices. Memoization stores the results of the minimum number of multiplications needed for a given chain length, converting an exponential problem into a polynomial one.

Advantages of Memoization

- **Efficiency**: Greatly improves the efficiency of algorithms by avoiding redundant calculations.

- **Simplicity**: Easy to implement in a recursive solution with minimal changes.

Considerations

- **Memory Usage**: While memoization accelerates computation, it can increase memory usage significantly, which might be a concern for space-constrained environments.

- **Problem Suitability**: Not all problems benefit from memoization. It's most effective when the problem has overlapping subproblems and a recursive structure.

Memoization is a cornerstone technique in optimizing recursive algorithms, particularly in the realm of Dynamic Programming. By understanding and applying memoization effectively, you can solve a wide range of complex problems more efficiently, both in terms of time and computational resources.

Greedy Algorithms

Greedy Algorithms are a class of algorithms that build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or that appears to be the best solution at the moment. This approach can lead to globally optimized solutions for some problems and only locally optimized solutions for others, depending on the problem's structure.

Key Concepts

1. **Local Optimum Choices**: Greedy algorithms make decisions from the given solution domain based on some local selection criterion.
2. **No Backtracking**: Once a choice is made, the algorithm never revisits or reverses this decision, which differentiates it from other techniques like backtracking or dynamic programming that might explore multiple possibilities before settling on a solution.

Runtime Complexity

- The time complexity of greedy algorithms varies significantly across different problems but is generally efficient for the problems they are applicable to. For example, the runtime for sorting algorithms can be $\mathcal{O}(n \log(n))$, Greedy Best First Search can operate in polynomial time, and constructing a Minimum Spanning Tree (MST) with algorithms like Prim's or Kruskal's can also be achieved efficiently.
- The efficiency of greedy algorithms often comes from their nature of making a single pass through the problem's data, making decisions that seem best at the moment without considering the future consequences in detail.

Spatial Complexity

Space complexity for greedy algorithms is typically lower than that for dynamic programming solutions because they do not need to store all the subproblem solutions. For example, in the case of algorithmic approaches for finding MSTs, the space complexity primarily depends on the representation of the graph (e.g., adjacency list or adjacency matrix) and the additional structures used (like priority queues in Prim's algorithm), which is often $\mathcal{O}(V)$ or $\mathcal{O}(E)$.

Applications and Examples

1. **Huffman Coding**: A compression algorithm that assigns variable length codes to input characters, with shorter codes for more frequent characters. This is a pure greedy approach.
2. **Activity Selection Problem**: Given a set of activities with their start and end times, the goal is to select the maximum number of activities that don't overlap. Greedy algorithms solve this by always picking the next activity that ends the earliest.
3. **Minimum Spanning Tree (MST)**: Algorithms like Kruskal's and Prim's algorithm are used to find the MST of a graph, which is a subset of the edges that connects all vertices in the graph with the minimum total edge weight.
4. **Dijkstra's Algorithm**: Used for finding the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.

Advantages of Greedy Algorithms

Greedy algorithms are generally more straightforward to conceptualize and can be more efficient in terms of runtime and space for the problems they are suitable for.

Considerations

A major consideration when using a greedy algorithm is whether a greedy choice will lead to an optimal solution. Greedy algorithms work best when every step is a choice that leads to an optimal solution, which is not guaranteed for all problems.

Greedy algorithms are powerful due to their simplicity and the efficiency with which they can solve certain classes of problems. They are particularly useful when a problem has a structure that guarantees that local optimal choices can lead to a global optimum, making them an essential tool in the algorithmic toolbox.

Graph Theory

Graph theory is a fundamental area of computer science and mathematics that involves the study of graphs, which are abstract models used to represent a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by vertices (or nodes), and the links that connect them are called edges. Within graph theory, two important traversal techniques are Breadth-First Search (BFS) and Depth-First Search (DFS). These techniques are pivotal for exploring nodes and edges of a graph and have varied applications, complexities, and characteristics.

Breadth-First Search (BFS)

BFS starts at a selected node (the source) and explores all of its nearest neighbors before moving to their neighbors, spreading outwards from the source in a wave-like manner. BFS is implemented using a queue to keep track of the next location to visit.

- **Functioning**: BFS explores vertices in the order of their distance from the source node, layer by layer. It ensures that for any reachable vertex v, BFS visits v only after visiting all vertices at a distance $d-1$ from the source, where d is the distance from the source to v.
- **Complexity**: The time complexity of BFS is $\mathcal{O}(V+E)$ for a graph represented using an adjacency list, where V is the number of vertices and E is the number of edges. The space complexity is $\mathcal{O}(V)$, as it needs to store a queue of vertices to explore.
- **Applications**: BFS is used in shortest path algorithms, algorithm to find connected components in an undirected graph, and in algorithms for computing the minimum spanning tree.

Depth-First Search (DFS)

DFS explores a graph by going as far into the graph as possible before backtracking to explore other paths. It's implemented using recursion or a stack to keep track of the exploration path.

- **Functioning**: DFS starts at the source node and explores as far as possible along each branch before backtracking. This means it will explore one neighbor of a node, then one neighbor of this neighbor, and so on, until it reaches an unexplored node or an endpoint.
- **Complexity**: The time complexity of DFS is also $\mathcal{O}(V+E)$ for a graph represented using an adjacency list. The space complexity, primarily due to the stack used for recursion, is $\mathcal{O}(V)$ in the worst case.
- **Applications**: DFS is utilized for topological sorting, detecting cycles in a graph, and finding strongly connected components.

Differences Between BFS and DFS

- **Strategy**: BFS explores neighbors first, making it suitable for finding the shortest path, while DFS dives deep into one neighbor before exploring others, useful for tasks that need to explore all possible paths.
- **Implementation**: BFS uses a queue to achieve its level-by-level exploration, whereas DFS uses a stack (either explicitly or implicitly through recursion) to keep track of the vertices to be explored.
- **Space Complexity**: In a sparse graph, both have similar space complexities, but their behavior might differ significantly in dense graphs or in their applications.

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

Rod Cutting Problem

The Rod Cutting Problem is a classic optimization problem that exemplifies the power of Dynamic Programming (DP). The problem statement is as follows:

'Given a rod of length n and a table of prices $p[i]$ for $i = 1, 2, \ldots, n$, determine the maximum revenue $r[n]$ obtainable by cutting up the rod and selling the pieces.'

Without Dynamic Programming

Without DP, a naive approach to solving the Rod Cutting Problem would involve checking all possible ways of cutting the rod and comparing the total revenue from each combination to find the maximum. This can be done using recursion, where the maximum revenue for a rod of length n is the maximum of $p[i] + r[n-i]$ for $i = 1, 2, \ldots, n$. However, this recursive approach recomputes solutions for the same subproblems many times and has an exponential time complexity of $\mathcal{O}(2^n)$, which is highly inefficient for larger values of n.

With Dynamic Programming

DP improves performance by systematically solving the subproblems only once and storing their solutions in a table from which we can reconstruct the final solution. This approach ensures that each subproblem is solved exactly once, eliminating the redundancy of the recursive approach.

The memoization strategy involves writing a recursive function that solves the problem for a rod of length n by dividing it into two parts at each step: a part of length i and a part of length $n-i$, for all i in $1, 2, \ldots, n$. Before making the recursive call to solve the problem for $n-i$, the function checks whether the value has already been computed. If it has, the function uses the stored value; if not, it computes and then stores the value.

Memoized Solution To The Rod Cutting Problem

Improvements with Memoization

By using memoization, the algorithm ensures that the work done for each subproblem of length k is not repeated. As a result, each subproblem is solved exactly once, and the results are reused, leading to significant performance improvements. We can see an implementation of this strategy below.

```python
def cut_rod_memoized(prices, n, revenue):
    if revenue[n] >= 0:
        return revenue[n]

    if n == 0:
        max_revenue = 0
    else:
        max_revenue = -float('inf')
        for i in range(1, n+1):
            max_revenue = max(max_revenue, prices[i] + cut_rod_memoized(prices, n-i, revenue))

    revenue[n] = max_revenue
    return max_revenue

def cut_rod(prices, n):
    revenue = [-float('inf')] * (n+1)
    return cut_rod_memoized(prices, n, revenue)

# Example usage:
prices = [0, 1, 5, 8, 9, 10, 17, 17, 20]  # Assuming 1-indexed prices array
rod_length = 8
print("Maximum Revenue:", cut_rod(prices, rod_length))

```

In this memoized version, `cut_rod_memoized` is the recursive function that computes the maximum revenue, with revenue serving as the memoization array. When `cut_rod` is called, it initializes this array with negative values to indicate that the subproblem solutions are initially uncomputed and then calls `cut_rod_memoized` to get the maximum revenue.

Runtime Complexity

With memoization, the runtime complexity improves to $\mathcal{O}(n^2)$ from the naive recursive approach's $\mathcal{O}(2^n)$. Although the memoized solution involves the same number of subproblems as the bottom-up approach, it differs by solving them in a top-down manner.

Spacial Complexity

The space complexity remains $\mathcal{O}(n)$ because we need to store the maximum revenue for each length up to n. However, because this is a recursive approach, there's also the added space complexity of the call stack, which in the worst case, can grow to $\mathcal{O}(n)$ as well.

Coin Changing Problem

The Coin Change Problem is a classic problem that asks:

'Given an unlimited supply of coins of given denominations, how many different ways can we make change for a particular amount of money?'

The problem can be solved efficiently using Dynamic Programming, specifically with the memoization technique.

Without Dynamic Programming

A naive approach might use brute-force recursion to try every combination of coins, which results in exponential time complexity due to the redundant calculation of subproblems.

With Dynamic Programming

Memoization avoids redundant calculations by storing the results of subproblems in some form of table for quick lookup. We define a recursive function that takes the amount to change and the number of coins as its arguments and returns the number of ways to make change.

Memoized Solution To The Coin Changing Problem

Improvements with Memoization

Memoization reduces the time complexity from exponential to polynomial. Each subproblem is uniquely defined by the remaining amount and the set of coins to be considered. Once computed, the result is stored. The solution to this problem using DP can be seen below.

```python
def count_coin_change_ways(coins, amount, index, memo):
    # Base cases
    if amount == 0:
        return 1  # One way to make change for 0, which is no coins
    if amount < 0 or index == len(coins):
        return 0  # No way to make change

    # Check the memo table to avoid re-computation
    if memo[amount][index] != -1:
        return memo[amount][index]

    # Recursive breakdown: include the coin vs exclude the coin
    # Include the coin: reduce the amount, keep the index same
    count_including_coin = count_coin_change_ways(coins, amount - coins[index], index, memo)

    # Exclude the coin: keep the amount same, go to next index
    count_excluding_coin = count_coin_change_ways(coins, amount, index + 1, memo)

    # Store the computed value in the memo table
    memo[amount][index] = count_including_coin + count_excluding_coin

    return memo[amount][index]

def coin_change(coins, amount):
    # Initialize the memo table with -1
    memo = [[-1 for _ in range(len(coins))] for _ in range(amount + 1)]
    return count_coin_change_ways(coins, amount, 0, memo)

# Example usage:
coins = [1, 2, 5]  # Coin denominations
amount = 5  # Amount to make change for
print("Number of ways to make change:", coin_change(coins, amount))

```

In this example, the function `count_coin_change_ways` uses memoization to store results of subproblems in a two-dimensional list memo. The `coin_change` function initializes this memo structure and starts the recursive process. Each entry `memo[amount][index]` represents the number of ways to make change for amount using coins from `coins[index]` onwards. The use of memoization ensures that each subproblem is calculated only once, thus improving the efficiency of the solution.

Runtime Complexity

With memoization, the time complexity is $\mathcal{O}(m * n)$, where m is the number of coin denominations and n is the amount to make change for. This complexity arises because each unique subproblem is solved only once.

Spatial Complexity

The space complexity for the memoized approach is also $\mathcal{O}(m * n)$ because the algorithm needs to store the solution for each subproblem. Additionally, the recursive implementation introduces a call stack depth which, in the worst case, can be $\mathcal{O}(n)$.

Knapsack Problem

The Knapsack Problem is a classic problem in combinatorial optimization. The problem can be described as follows:

'You are given a set of n items, each with a weight $w[i]$ and a value $v[i]$, and a knapsack that can carry a maximum weight W. The goal is to determine the maximum value that the knapsack can carry.'

Without Dynamic Programming

A brute-force solution would examine all subsets of items to find the maximum value that fits in the knapsack. This approach has a time complexity of $\mathcal{O}(2^n)$, as there are 2^n possible combinations of items.

With Dynamic Programming

We define a recursive function that computes the maximum value for a given capacity and number of items. We store the results of these computations in a two-dimensional array where one dimension represents the remaining capacity and the other represents the number of items considered.

Memoized Solution To The Knapsack Problem

Improvements with Memoization

Memoization brings down the complexity significantly. Instead of solving the exponential number of subproblems, we only solve $\mathcal{O}(n * W)$ subproblems since there are at most n items and W capacity constraints to consider. The improved solution to this problem using DP can be seen below.

```python
def knapsack_memoized(values, weights, capacity, index, memo):
    # Base Case: If no items are left or capacity is 0.
    if index == len(values) or capacity == 0:
        return 0

    # If the result is already in the memo table then return that value.
    if memo[index][capacity] != -1:
        return memo[index][capacity]

    # If the weight of the item is more than the knapsack's capacity, skip this item.
    if weights[index] > capacity:
        memo[index][capacity] = knapsack_memoized(values, weights, capacity, index+1, memo)
    else:
        # Recursive call to choose the item and not to choose the item.
        value_including_item = values[index] + knapsack_memoized(values, weights, capacity - weights[index], index+1, memo)
        value_excluding_item = knapsack_memoized(values, weights, capacity, index+1, memo)

        # Store the maximum of including or excluding the current item.
        memo[index][capacity] = max(value_including_item, value_excluding_item)

    return memo[index][capacity]

def knapsack(values, weights, capacity):
    # Initialize the memo table with -1
    memo = [[-1 for _ in range(capacity + 1)] for _ in range(len(values))]
    return knapsack_memoized(values, weights, capacity, 0, memo)

# Example usage:
values = [60, 100, 120]  # The values of the items
weights = [10, 20, 30]   # The weight of the items
capacity = 50            # The maximum capacity of the knapsack
print("Maximum value in Knapsack =", knapsack(values, weights, capacity))

```

In this code, `knapsack_memoized` is the recursive function that computes the maximum value, and memo is a 2D array that serves as the memoization table. When `knapsack` is called, it initializes this table and starts the recursion process. The memoization ensures we do not recompute the value for any state (defined by index and capacity), hence significantly optimizing the brute-force approach.

Runtime Complexity

With memoization, the runtime complexity becomes $\mathcal{O}(n * W)$ where n is the number of items and W is the knapsack capacity. This is because each state of the DP is defined by the current item and the remaining

capacity, and we solve each state only once.

Spatial Complexity

The space complexity is $\mathcal{O}(n * W)$ because we store the result for each item-capacity combination in a memoization table. If we're using recursion, we also need to consider the call stack, which can add up to $\mathcal{O}(n)$ in space in the worst case.

Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) problem is a classic problem in computer science. The goal is to find the longest subsequence present in two sequences (which could be strings, arrays, etc.). A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous (as opposed to a substring) in both sequences.

Without Dynamic Programming

A brute-force solution to the LCS problem would examine all subsequences of both sequences and check for the longest matching one. This approach would have a time complexity of $\mathcal{O}(2^n * 2^m)$ for two sequences of length n and m, as it involves generating all subsequences for both sequences.

With Dynamic Programming

Memoization improves the solution by avoiding the recalculation of the LCS length for the same subproblem. In the context of LCS, a subproblem is defined as finding the LCS length of prefixes of the sequences.

We define a recursive function that computes the LCS length for two given indices in the sequences, one for each sequence. We then store the results in a two-dimensional array (often called a memoization table) so that each subproblem is computed only once.

Memoized Solution To The Longest Common Subsequence Problem

Improvements with Memoization

Memoization reduces the time complexity from exponential to polynomial. Each subproblem is uniquely defined by the indices within the two sequences and, once computed, will not be recomputed. The improved solution with memoization can be seen below.

```python
def lcs_memoized(seq1, seq2, i, j, memo):
    if i == 0 or j == 0:
        return 0  # Base case: If either sequence is empty

    if memo[i][j] != -1:
        return memo[i][j]  # Return memoized result if already computed

    if seq1[i-1] == seq2[j-1]:
        # If characters match, 1 + LCS of remaining sequences
        memo[i][j] = 1 + lcs_memoized(seq1, seq2, i-1, j-1, memo)
    else:
        # If no match, max of LCS of seq1 minus current char and seq2 minus current char
        memo[i][j] = max(lcs_memoized(seq1, seq2, i, j-1, memo), lcs_memoized(seq1, seq2, i-1, j, memo))
    return memo[i][j]

def lcs(seq1, seq2):
    memo = [[-1 for _ in range(len(seq2) + 1)] for _ in range(len(seq1) + 1)]
    return lcs_memoized(seq1, seq2, len(seq1), len(seq2), memo)

# Example usage:
seq1 = "AGGTAB"
seq2 = "GXTXAYB"
print("Length of LCS is", lcs(seq1, seq2))

```

In this code, `lcs_memoized` is the recursive function that computes the length of the LCS, and memo is a 2D array that serves as the memoization table. The function `lcs` initializes the memo table and kicks off the recursion. Each entry `memo[i][j]` represents the length of the LCS for the first i characters of `seq1` and the first j characters of `seq2`. Using memoization ensures that each subproblem is only solved once, greatly

optimizing the naive recursive approach.

Runtime Complexity

With memoization, the runtime complexity is $\mathcal{O}(n * m)$, where n is the length of the first sequence and m is the length of the second sequence. This is because there are $n * m$ possible states and each state is solved only once.

Spatial Complexity

The space complexity is $\mathcal{O}(n*m)$ due to the memoization table that stores the solution for each subproblem. If recursion is used, the call stack space should also be taken into account, which could add up to $\mathcal{O}(n + m)$ in space in the worst case.

The Fibonacci Sequence

The Fibonacci sequence is a famous series in mathematics where each number is the sum of the two preceding ones, typically starting with 0 and 1. That is, $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n > 1$.

Without Dynamic Programming

A naive approach to computing the nth Fibonacci number might involve a simple recursive function. This function would call itself to obtain the two preceding Fibonacci numbers until reaching the base cases of $F(0)$ and $F(1)$. This approach does repeated work and has an exponential time complexity of approximately $\mathcal{O}(2^n)$, due to the growth of the recursion tree exponentially at each level.

With Dynamic Programming

Using memoization drastically reduces the number of function calls, thus improving the performance from exponential to linear time complexity.

Memoized Solution To The Fibonacci Sequence

Improvements with Memoization

Using memoization drastically reduces the number of function calls, thus improving the performance from exponential to linear time complexity. This improvement can be seen below.

```python
def fib_memoized(n, memo):
    if n in memo:
        return memo[n]  # Return the cached result

    if n <= 1:
        return n  # Base cases

    # Recursive calls and store the results in memo
    memo[n] = fib_memoized(n-1, memo) + fib_memoized(n-2, memo)
    return memo[n]

def fibonacci(n):
    memo = {}
    return fib_memoized(n, memo)

# Example usage:
n = 10
print(f"Fibonacci number at position {n} is {fibonacci(n)}")

```

In this implementation, `fib_memoized` is a helper function that uses a dictionary named `memo` to store the Fibonacci numbers that have already been computed. The fibonacci function initializes this memoization store and calls `fib_memoized`. This approach uses memoization to ensure that each Fibonacci number is calculated once, yielding a significant improvement in performance for large n compared to the naive recursive

approach.

Runtime Complexity

The time complexity with memoization is $\mathcal{O}(n)$ because we compute each Fibonacci number once and then retrieve the result from the cache for subsequent calls.

Spatial Complexity

The space complexity is also $\mathcal{O}(n)$ because we need to store the result for each of the n Fibonacci numbers. This storage is typically done in an array or a hash map.

Exam 4 Notes

Dijkstra's Algorithm

Dijkstra's Algorithm is a classic algorithm in computer science, used to find the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights. This algorithm is particularly useful in GPS networks to find the shortest path efficiently.

Algorithm Overview

The essence of Dijkstra's Algorithm lies in its use of a priority queue to repeatedly select the vertex with the smallest tentative distance from the source. The algorithm maintains several key structures during execution:

- A set, often referred to as `visited`, to keep track of vertices whose minimum distance from the source has already been determined.
- An array or dictionary `dist`, where `dist[v]` holds the shortest found distance from the source to vertex `v`.
- A priority queue that prioritizes vertices based on their tentative distance from the source, helping to explore the most promising vertex next.

Operation Per Iteration

During each iteration of Dijkstra's Algorithm, the following steps are executed:

1. Select the vertex `u` with the smallest tentative distance (`dist[u]`) from the priority queue that has not been visited yet.
2. Mark vertex `u` as visited. This means that the shortest path to `u` has been permanently determined.
3. For each neighboring vertex `v` of `u` that is not visited:
   (a) Calculate the alternative path distance to `v`, which is `dist[u]` plus the weight of the edge from `u` to `v`.
   (b) If this alternative distance is less than the currently known distance `dist[v]`, update `dist[v]` with the smaller distance and insert `v` into the priority queue with its updated distance.

Vertex Exploration Order

The order in which vertices are explored in Dijkstra's Algorithm is determined by the priority queue which prioritizes vertices based on their distance from the source. This structure ensures that at any step, the vertex with the lowest known distance is explored next, implementing a greedy strategy to always move towards the closest unvisited vertex. This property is crucial for guaranteeing that the found paths are the shortest.

Complexity

The time complexity of Dijkstra's Algorithm can vary depending on the implementation:

- Using a binary heap for the priority queue results in a complexity of $\mathcal{O}((V + E)\log(V))$, where V is the number of vertices and E is the number of edges.
- If a Fibonacci heap is used, the complexity can be improved to $\mathcal{O}(E + V\log(V))$.

This makes Dijkstra's algorithm very efficient for graphs with large numbers of vertices but relatively fewer edges.

Minimum Spanning Trees (MST)

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimal possible total edge weight. This concept is crucial in networks like telecommunication, where the goal is to lay out cables at the minimum possible cost.

Properties of MST

The properties of a Minimum Spanning Tree include:

- An MST for a graph with V vertices has exactly $V - 1$ edges.
- If all weights are distinct, the MST is unique; however, if weights are not distinct, multiple MSTs may exist.
- The total weight of all the edges in the MST is the minimum among all possible spanning trees of the graph.
- MSTs are acyclic and connect all vertices in the graph.

Kruskal's Algorithm

Kruskal's Algorithm is a popular method to find the MST of a graph. It follows a greedy approach and is typically easier to implement than Prim's algorithm. The steps are as follows:

1. Sort all the edges in non-decreasing order of their weight.
2. Initialize a forest F as a set of trees with each vertex in the graph as a separate tree.
3. For each sorted edge (u, v):
   (a) If u and v are in different trees (i.e., adding (u, v) does not form a cycle),
   (b) Add (u, v) to F, and merge the two trees in F into one (a union-find data structure is commonly used for this step).
4. The process continues until F contains exactly one tree, which is the MST.

Complexity of Kruskal's Algorithm

The time complexity of Kruskal's Algorithm mainly depends on the edge sorting step and the union-find operations:

- Sorting the edges takes $\mathcal{O}(E \log(E))$ time.
- Each union-find operation (both union and find) can be optimized to $\mathcal{O}(\log(V))$ using path compression and union by rank, making the overall complexity of managing the forest $\mathcal{O}(E \log(V))$.

Therefore, the total time complexity of Kruskal's Algorithm is $\mathcal{O}(E \log(E))$, which is effective for sparse graphs.

Depth First Search (DFS) and Breadth First Search (BFS)

Depth First Search (DFS) and Breadth First Search (BFS) are two fundamental algorithms used for exploring vertices and edges of a graph. These methods are pivotal for numerous applications in computer science, from pathfinding to analyzing networks.

Depth First Search (DFS)

DFS explores as far as possible along each branch before backing up. Here's how it works:

1. Start at the chosen vertex, marking it as visited.
2. Recursively visit any adjacent vertex that has not been visited yet, marking it as visited when you visit it.
3. This recursive visiting continues until all vertices connected by a path to the starting vertex are visited.

DFS can be implemented using a stack, either through the function call stack via recursion or an explicit stack data structure.

Breadth First Search (BFS)

BFS explores the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. Here's the procedure:

1. Start at the chosen vertex, marking it as visited and enqueue it.
2. While the queue is not empty:
   (a) Dequeue the next vertex from the front of the queue.
   (b) Visit all its adjacent vertices, mark them as visited, and enqueue any vertex that has not been visited yet.

BFS uses a queue data structure to ensure that vertices are explored in the order they are reached.

Strongly Connected Components (SCC)

A Strongly Connected Component (SCC) of a directed graph is a maximal subset of vertices such that every vertex in the subset is reachable from every other vertex in the subset.

Properties of SCC

- In a directed graph, an SCC forms a cycle which means there is a path from each vertex to every other vertex in the same component.
- The union of all SCCs of a graph forms the entire graph.
- SCCs are the "meta-nodes" in the Condensed Graph of the original graph where each SCC is a node, and edges between SCCs represent paths in the original graph.

Finding SCCs

The common algorithms for finding all SCCs in a graph are:

1. Tarjan's Algorithm: It uses DFS to locate SCCs and can find all SCCs in one go, marking each strongly connected component as it goes.
2. Kosaraju's Algorithm: This method involves two passes of DFS. The first pass orders the vertices in decreasing finish time. In the second pass, the graph is reversed and DFS is applied in the order determined in the first pass.

Both algorithms run in linear time, $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges.

Reconstructing Graphs from Strongly Connected Components (SCC)

Given the SCCs of a directed graph, one might wonder about the structure of the original graph. Particularly, questions often arise regarding the minimum and maximum number of edges that the graph could potentially have based on its SCCs.

Understanding SCCs in Graph Reconstruction

Example: Consider a directed graph with 6 nodes divided into three maximal SCCs: {1,2}, {3,4}, {5,6}.

Graph Reconstruction from SCCs

Given these SCCs, we know each SCC forms a subgraph where every node is reachable from every other node within the same SCC. The challenge lies in determining the possible connections between these SCCs and the edges within each SCC.

Minimum Number of Edges

The minimum number of edges in the graph is achieved by:

- Including the minimum number of edges required to keep each SCC strongly connected.
- Not adding any additional edges between these SCCs.

Calculation:

- Each SCC with 2 nodes requires at least 2 edges to be strongly connected (forming a cycle).
- Therefore, for SCCs {1,2}, {3,4}, and {5,6}, we need at least 2+2+2 = 6 edges.

Thus, the minimum number of edges is 6.

Maximum Number of Edges

The maximum number of edges is computed by:

- Including all possible edges within each SCC to maintain strong connectivity.
- Adding all possible directed edges between different SCCs.

Calculation:

- Within each SCC of 2 nodes, we can have 2 direct edges (one in each direction), making it 4 possible edges (full connectivity).
- Between each pair of SCCs (say, between {1,2} and {3,4}), there can be 4 directed edges (1 to 3, 1 to 4, 2 to 3, 2 to 4).

- There are 3 pairs of such SCCs ({1,2} & {3,4}, {1,2} & {5,6}, and {3,4} & {5,6}), contributing 4*3 = 12 inter-SCC edges.
- Total edges within SCCs: 4+4+4 = 12.

Therefore, the maximum number of edges is 12 + 12 = 24.

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.