DAA Theory Notes — Midterm Exam Preparation

Design and Analysis of Algorithms

UNIT I — Introduction to Algorithms

---

1. Algorithm — Definition and Properties

1. An algorithm is a finite set of well-defined, unambiguous instructions that takes valid input and produces the correct output in a finite amount of time.
2. **Input** — An algorithm must have zero or more clearly defined inputs that are provided to it before execution begins.
3. **Output** — The algorithm must produce at least one output that is provably correct for any valid input given to it.
4. **Finiteness** — The algorithm must always terminate after a finite and countable number of steps and cannot run indefinitely.
5. **Definiteness** — Every single instruction must be clear, precise, and completely unambiguous so there is no room for misinterpretation.
6. **Effectiveness** — Every instruction must be simple and basic enough to be carried out exactly in a finite amount of time.

---

2. Analysis Framework

1. The analysis framework provides a standard way to measure and compare the efficiency of algorithms in terms of time and memory as input size n grows.
2. **Time Complexity** — measures how the number of basic operations such as comparisons and assignments grows as a function of input size n.
3. **Space Complexity** — measures the total amount of memory the algorithm requires during its complete execution.
4. **Best Case** — the minimum possible time the algorithm takes when given the most favorable possible input, for example finding the target at the first position in search.
5. **Worst Case** — the maximum time taken for the most unfavorable input and is most commonly used because it gives a guaranteed upper bound on performance.
6. **Average Case** — the expected running time calculated over all possible inputs and is more realistic but harder to compute mathematically.

---

3. Asymptotic Notations — Overview

1. Asymptotic notations are mathematical tools used to describe the growth rate of an algorithm's running time as input size n approaches infinity.
2. They ignore constant factors and lower-order terms because only the dominant term determines behavior for large inputs.
3. The three main notations are Big O (upper bound), Omega (lower bound), and Theta (tight bound).
4. As an analogy — O is like $\leqslant$ meaning at most this fast, Θ is like $=$ meaning exactly this fast, and Ω is like $\geqslant$ meaning at least this fast.
5. Common growth rates in increasing order of complexity: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$.

---

4. Big O Notation — O(g(n)) — Asymptotic Upper Bound

1. Big O gives the **upper bound** on the running time, meaning it represents the worst case growth rate of the algorithm.
2. **Formal Definition**: $f(n) = O(g(n))$ if there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0 \leqslant f(n) \leqslant c \cdot g(n)}$ for all $n \geqslant n_0$.
3. It means f(n) grows no faster than g(n) beyond a certain input size n_0, so g(n) is an upper limit.
4. **Example**: $f(n) = n^2 + 5n = O(n^2)$ because $n^2 + 5n \leqslant 2n^2$ for all $n \geqslant 5$, where $c = 2$ and $n_0 = 5$.
5. **Proof Example**: $T(n) = 15n^3 + n^2 + 4 = O(n^3)$. Choose $c = 20$ and $n_0 = 1$. Then $15n^3 + n^2 + 4 \leqslant 20n^3$ holds for all $n \geqslant 1$.

---

5. Omega Notation — Ω(g(n)) — Asymptotic Lower Bound

1. Omega gives the **lower bound** on the running time, meaning it represents the best case growth rate of the algorithm.
2. **Formal Definition**: $f(n) = \Omega(g(n))$ if there exist positive constants **c** and $\mathbf{n_0}$ such that $\mathbf{0 \leqslant c \cdot g(n) \leqslant f(n)}$ for all $n \geqslant n_0$.
3. It means f(n) grows no slower than g(n), so the algorithm will always take at least this much time.
4. **Example**: $f(n) = n^2 + 5n = \Omega(n^2)$ because $n^2 + 5n \geqslant 1 \cdot n^2$ for all $n \geqslant 1$, where $c = 1$ and $n_0 = 1$.

5. **Proof Example:** $T(n) = 15n^3 + n^2 + 4 = \Omega(n^3)$. Choose $c = 15$ and $n_0 = 1$. Then $15n^3 \leqslant 15n^3 + n^2 + 4$ holds for all $n \geqslant 1$.

---

6. Theta Notation — Θ(g(n)) — Asymptotic Tight Bound

1. Theta gives the **tight bound**, meaning the algorithm's running time grows at exactly the rate of $g(n)$, neither faster nor slower.
2. **Formal Definition:** $f(n) = \Theta(g(n))$ if there exist positive constants $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{n_0}$ such that $\mathbf{c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)}$ for all $n \geqslant n_0$.
3. $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$ — it must satisfy both upper and lower bound simultaneously.
4. **Example:** $f(n) = n^2 + 5n = \Theta(n^2)$ because it is both $O(n^2)$ and $\Omega(n^2)$ — the n^2 term dominates.
5. If $T(n)$ is a polynomial of degree k then $T(n) = \Theta(n^k)$, meaning Theta always picks the highest degree term as the dominant term.

---

7. Recurrence Relations

1. A recurrence relation is a mathematical equation that expresses the running time $T(n)$ of a recursive algorithm in terms of its running time on smaller inputs.
2. It is formed naturally when a recursive algorithm divides a problem of size n into smaller subproblems — the total cost is the cost of the subproblems plus the cost of combining them.
3. **Example:** Merge Sort divides n elements into 2 halves and takes $O(n)$ to merge them, giving the recurrence $T(n) = 2T(n/2) + n$.
4. **Base Case:** Every recurrence must have a base case such as $T(1) = 1$ or $T(0) = 0$ which stops the recursion and gives the starting value for the induction.
5. The two main methods to solve recurrence relations are the **Master Theorem** (for divide and conquer recurrences of form $T(n) = aT(n/b) + f(n)$) and the **Substitution Method** (for any form using mathematical induction).

---

8. Master Theorem

1. The Master Theorem provides a direct formula to solve recurrences of the form $\mathbf{T(n) = aT(n/b) + f(n)}$ where $a \geqslant 1$ and $b \geqslant 2$.
2. Here **a** is the number of subproblems, **b** is the factor by which input size is divided each time, and $\mathbf{f(n) = \Theta(n^d)}$ is the cost of work done outside the recursion.

3. Compare **a** with $\mathbf{b^d}$ to determine which of the three cases applies:
   - **Case 1**: If $a < b^d$ then $T(n) = \mathbf{\Theta(n^d)}$ — the work done outside recursion dominates the total cost.
   - **Case 2**: If $a = b^d$ then $T(n) = \mathbf{\Theta(n^d \log n)}$ — both costs are equal so a log factor is added to the result.
   - **Case 3**: If $a > b^d$ then $T(n) = \mathbf{\Theta(n^{(\log_b a)})}$ — the cost of the recursive subproblems dominates.
4. **Example**: $T(n) = 2T(n/2) + n \rightarrow a=2, b=2, d=1, b^d=2, a = b^d \rightarrow$ **Case 2 → $T(n) = \Theta(n \log n)$**.
5. **Example**: $T(n) = 3T(n/2) + n \rightarrow a=3, b=2, d=1, b^d=2, a > b^d \rightarrow$ **Case 3 → $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$**.

---

9. Substitution Method

1. The substitution method solves recurrences by first **guessing the form of the solution** and then proving that guess is correct using mathematical induction.
2. **Step 1 — Guess**: Make an educated guess about the solution form, for example assume $T(n) = O(n \log n)$.
3. **Step 2 — Inductive Hypothesis**: Assume the guess holds true for all smaller inputs, for example assume $T(n/2) \leqslant c \cdot (n/2) \cdot \log(n/2)$.
4. **Step 3 — Substitute**: Plug the inductive hypothesis back into the original recurrence and simplify to check whether the inequality holds.
5. **Step 4 — Find Constants**: Identify specific values of constants c and n_0 that make the inequality true to complete the formal inductive proof.

---

10. Linear Search (Sequential Search)

1. Linear search scans every element in the array one by one from left to right until the target element is found or the entire array is exhausted.
2. It works on both sorted and unsorted arrays because it does not assume or require any particular order of elements.
3. **Best Case**: $O(1)$ — the target element is found at the very first position in the array.
4. **Worst Case**: $O(n)$ — the target is either at the last position or completely absent, requiring all n comparisons to be made.
5. Linear search is simple to implement but inefficient for large datasets compared to binary search.

```
Algorithm Search(array, target, size)
{
  for i = 1 to size do
  {
    if (array[i] == target)
    { print "Target found at position i"; break; }
  }
  if (i >= size)
    print "Target not found"
}
```

---

11. Binary Search

1. Binary search works only on sorted arrays — it finds the target by repeatedly dividing the search space into two halves.
2. It compares the target with the middle element — if they match it returns the position, if the target is smaller it searches only the left half, if larger it searches only the right half.
3. **Best Case**: O(1) — the target happens to be the middle element and is found in the very first comparison.
4. **Worst Case**: O(log n) — the search space is halved at each step so at most $\log_2 n$ comparisons are ever needed.
5. Binary search is significantly faster than linear search for large sorted arrays because it eliminates half the remaining elements at every single step.

```
Algorithm binary_search(a[], low, high, key)
{
  while (low <= high)
  {
    mid = (low + high) / 2
    if (a[mid] == key) return mid
    else if (key < a[mid]) high = mid - 1
    else low = mid + 1
  }
  return -1
}
```

---

12. Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Input requirement	Works on both sorted and unsorted arrays without any precondition	Requires the array to be fully sorted before searching can begin
Best case complexity	O(1) when the element happens to be at the very first position	O(1) when the element is at the middle position on the first comparison
Worst case complexity	O(n) because every element may need to be checked in the worst case	O(log n) because the search space is halved at each step
Method used	Scans elements one by one in a sequential manner from start to end	Divides the search space in half at every comparison to zoom in on the target
Best suited for	Small datasets or unsorted data where sorting overhead is not worth it	Large sorted datasets where fast lookup performance is required

---

UNIT II — Divide and Conquer and Greedy Strategy

---

1. Divide and Conquer — Introduction and Principle

1. Divide and Conquer is an algorithm design strategy that breaks a large problem into smaller independent subproblems, solves each recursively, and combines the results into the final answer.
2. It follows three steps — **Divide** the problem into smaller instances, **Conquer** each instance recursively, and **Combine** all solutions into the final answer.
3. It always produces a recursive algorithm and the time complexity is naturally expressed as a recurrence relation of the form $T(n) = aT(n/b) + f(n)$.
4. The divide and conquer approach works best when subproblems are independent and do not overlap — if they overlap and share subproblems then dynamic programming is preferred.
5. Well-known examples include Merge Sort, Quick Sort, and Binary Search — all divide the input into parts and solve each part recursively.

---

2. Control Abstraction of Divide and Conquer

1. Control abstraction is a general template that captures the structure common to all Divide and Conquer algorithms in a single reusable pseudocode form.
2. It first checks if the problem P is **SMALL** enough to be solved directly — if yes it calls the base case solver S(P) and returns immediately.
3. If the problem is not small enough, it divides P into smaller subproblems p1, p2, ..., pk.
4. DANDC is applied recursively and independently to each subproblem until all are small enough to solve directly.
5. The COMBINE function merges all individual solutions into one final correct solution and returns it.

```
DANDC(P)
{
  if SMALL(P) then return S(P);
  else
  {
    divide P into p1, p2, ..., pk
    apply DANDC to each
    return COMBINE(DANDC(p1), ..., DANDC(pk))
  }
}
```

---

3. Merge Sort

1. Merge Sort is a divide and conquer sorting algorithm that splits the array into two equal halves, recursively sorts each half, and merges them back in sorted order.
2. **Divide** — Split the array at the midpoint into two halves, which takes $\Theta(1)$ constant time.
3. **Conquer** — Recursively sort each half independently, contributing $2T(n/2)$ to the overall recurrence.
4. **Combine** — Merge the two sorted halves back into a single sorted array, which takes $\Theta(n)$ linear time.
5. **Time Complexity**: Best, Worst, and Average cases are all $\mathbf{\Theta(n \log n)}$. Space complexity is $O(n)$ because extra memory is needed for the merge operation.

Recurrence: $T(n) = 2T(n/2) + n \rightarrow$ Case 2 of Master Theorem $\rightarrow \mathbf{T(n) = \Theta(n \log n)}$

```
Algorithm MergeSort(a, low, high)
{
  if (low < high)
  {
    mid = (low + high) / 2
    MergeSort(a, low, mid)
    MergeSort(a, mid+1, high)
    Merge(a, low, mid, high)
  }
}
```

---

4. Quick Sort

1. Quick Sort selects a pivot element, partitions the array so all elements smaller than the pivot go left and all larger ones go right, then recursively sorts both sides.
2. **Divide** — Select the pivot and partition the array into two groups around it, which takes O(n) linear time.
3. **Conquer** — Recursively apply Quick Sort on the left partition and the right partition independently.
4. **Combine** — No merging step is needed because the pivot is already in its final correct position after partitioning.
5. **Best/Average Case:** $\Theta(n \log n)$. **Worst Case:** $\Theta(n^2)$ — occurs when the pivot is always the smallest or largest element, such as in an already sorted array.

---

5. Merge Sort vs Quick Sort

Feature	Merge Sort	Quick Sort
Best Case	$\Theta(n \log n)$ always regardless of the input arrangement	$\Theta(n \log n)$ when the pivot consistently divides the array into roughly equal halves
Worst Case	$\Theta(n \log n)$ even in the worst case because the split is always equal	$\Theta(n^2)$ when the pivot is always the smallest or largest element causing severely unequal partitions
Extra Space	Requires O(n) additional memory for the temporary arrays used during merging	Requires only O(log n) stack space making it effectively in-place for the array data

Feature	Merge Sort	Quick Sort
Stability	Stable sort meaning equal elements always maintain their original relative order	Not a stable sort because swapping during partitioning may change the relative order of equal elements
Best suited for	Linked lists and external sorting where random access is expensive or unavailable	Arrays where in-place sorting and good average-case performance are the main priorities

---

6. Greedy Strategy — Introduction and Principle

1. A greedy algorithm solves optimization problems by making the locally best choice at each step without ever reconsidering or undoing any previous decision.
2. At each stage the algorithm selects the most attractive available option based on some selection criterion such as maximum profit ratio or minimum cost.
3. A **feasible solution** is any subset of the input that satisfies all the constraints of the problem, even if it is not the optimal one.
4. An **optimal solution** is the feasible solution that maximizes or minimizes the objective function — it is the best possible answer to the problem.
5. Greedy does not always guarantee a globally optimal solution for every problem, but for specific problems like Fractional Knapsack and Huffman Encoding it always produces the correct optimal answer.

---

7. Control Abstraction of Greedy Method

1. The Greedy control abstraction is a general template that describes the common iterative structure of all greedy algorithms in one reusable pseudocode.
2. **Select(a)** — picks the next best input element from the remaining candidates based on the greedy selection criterion such as highest ratio or lowest cost.
3. **Feasible(solution, x)** — checks whether including element x in the current partial solution still satisfies all problem constraints without violating any rule.
4. **Union(solution, x)** — adds element x to the current solution only if it passes the feasibility check successfully.
5. The loop continues until all candidate inputs have been considered and the final completed solution is returned.

```
Algorithm Greedy(a, n)
{
  solution = 0;
  for i = 1 to n do
  {
    x = Select(a);
    if Feasible(solution, x) then
      solution = Union(solution, x);
  }
  return solution;
}
```

---

8. Fractional Knapsack Problem (Greedy)

1. Given a knapsack of weight capacity m and n items each with profit p_i and weight w_i, the goal is to maximize total profit where fractional amounts of items are allowed.
2. **Greedy Strategy** — Sort all items by their profit-to-weight ratio (p_i/w_i) in decreasing order and always pick the item with the highest ratio first.
3. If the full item fits within the remaining capacity it is taken completely; if not, a fraction of it is taken to fill exactly the remaining space.
4. **Objective**: Maximize $\Sigma p_i x_i$ where $0 \leqslant x_i \leqslant 1$, subject to the capacity constraint $\Sigma w_i x_i \leqslant m$.
5. **Time Complexity**: $O(n \log n)$ because of the sorting step. This greedy approach always gives the globally optimal solution for the fractional version of the problem.

---

9. Job Sequencing with Deadlines (Greedy)

1. Given n jobs each with a profit p_i and a deadline d_i, select a subset of jobs to maximize total profit where each job takes exactly one unit of time and must finish by its deadline.
2. **Greedy Strategy** — Sort all jobs by profit in decreasing order and for each job assign it to the latest available time slot that is still within its deadline.
3. A **feasible solution** is any subset of jobs where each selected job can be scheduled to finish within its given deadline without conflict.
4. An **optimal solution** is the feasible solution with the maximum total profit among all possible feasible subsets.
5. **Time Complexity**: $O(n^2)$ in the worst case because the outer loop runs n times and finding an available slot within the deadline can take $O(k)$ per job.

Steps:

1. Sort all jobs by profit in decreasing order.
2. For each job, find the latest available empty slot that is ⩽ its deadline.
3. If such a slot is found assign the job there; otherwise discard the job entirely.

---

10. Huffman Encoding (Greedy)

1. Huffman coding is a greedy lossless data compression algorithm that assigns shorter binary codes to more frequent characters and longer codes to less frequent ones to minimize total bit length.
2. Create a leaf node for each character with its frequency as its weight and insert all nodes into a min-priority queue ordered by frequency.
3. Repeatedly extract the two nodes with the lowest frequencies, create a new internal parent node whose frequency equals their combined sum, and reinsert the parent back into the queue.
4. Repeat step 3 until only one node remains in the queue — this single remaining node becomes the root of the complete Huffman tree.
5. Assign 0 to every left branch and 1 to every right branch — the path from the root to any leaf node spells out that character's unique prefix-free binary code.

Time Complexity: O(n log n).

---

11. Divide and Conquer vs Greedy

Feature	Divide and Conquer	Greedy
Core approach	Divides the problem into independent subproblems, solves each recursively, and combines all results into the final answer	Makes the locally best choice at each step without breaking the problem into recursive subproblems
Subproblem solving	Solves all subproblems recursively and independently before combining their solutions together	Makes exactly one greedy decision per step and does not revisit or undo any previous choice
Reconsideration	Reconsiders all subproblem results during the combine phase to produce the globally correct final answer	Never reconsiders previous decisions — once a greedy choice is made it is permanently fixed
Optimality guarantee	Always produces the globally optimal solution by combining results of all subproblems	Not always globally optimal for every problem but gives optimal results for

Feature	Divide and Conquer	Greedy
		specific ones like Knapsack and Huffman
Examples	Merge Sort, Quick Sort, and Binary Search are classic divide and conquer algorithms	Fractional Knapsack, Huffman Encoding, and Job Sequencing are classic greedy algorithms

UNIT III — Graphs, Symbol Tables, and AVL Trees

1. Graph — Basic Terminology

1. A **graph G(V, E)** is a mathematical structure consisting of a finite nonempty set of vertices V and a finite set of edges E that connect pairs of vertices.
2. An **undirected graph** has edges with no direction — edge (u, v) is the same as edge (v, u) and represents a two-way bidirectional connection.
3. A **directed graph (digraph)** has edges with a specific direction — edge (u, v) goes only from u to v and is not the same as the reverse edge (v, u).
4. The **degree** of a vertex is the number of edges connected to it. In directed graphs, **in-degree** is the count of incoming edges and **out-degree** is the count of outgoing edges.
5. A **complete undirected graph** with n vertices has exactly n(n-1)/2 edges, while a complete directed graph has n(n-1) directed edges.

2. More Graph Terminology

1. A **path** is a sequence of vertices where each consecutive pair is connected by an edge, and the **length** of the path is the total number of edges in it.
2. A **simple path** is one where all vertices in the sequence are distinct — no vertex appears more than once along the path.
3. A **cycle** is a closed path where the starting vertex and the ending vertex are the same vertex, forming a loop.
4. A **connected graph** is one where there exists at least one valid path between every pair of distinct vertices in the graph.
5. A **connected component** is a maximal connected subgraph — the largest possible set of vertices and edges that forms a connected graph on its own within a larger disconnected graph.

3. Graph Topologies

1. **Linear / Path Graph** — Vertices are arranged in a straight sequential chain where each vertex connects only to the next one. It has exactly n-1 edges for n vertices and is used to represent simple sequential pipelines or linked structures.
2. **Ring / Cycle Graph** — Vertices form a closed circular loop where each vertex connects to exactly two neighbors and the last vertex connects back to the first, creating a cycle. It is used in token ring networks and circular buffer systems.
3. **Star Graph** — One central hub vertex connects directly to all other outer vertices while the outer vertices have no connections among themselves. The hub is a single point of failure — if it goes down the entire network disconnects.
4. **Tree Graph** — A connected acyclic graph with exactly n-1 edges for n vertices and no cycles of any kind. It represents hierarchical relationships such as file directory systems, organization charts, and compiler parse trees.
5. **Complete Graph** — Every vertex is directly connected to every other vertex in the graph by a unique edge. A complete undirected graph with n vertices has exactly n(n-1)/2 edges and represents the maximum possible connectivity.
6. **Bipartite Graph** — Vertices are divided into two distinct non-overlapping sets where edges only connect vertices from one set to vertices in the other set and never within the same set. It is used in assignment and matching problems like allocating jobs to workers.
7. **Directed Acyclic Graph (DAG)** — A directed graph that contains absolutely no directed cycles, meaning it is impossible to start at any vertex and return to it by following directed edges. It is the essential structure for topological sorting and dependency resolution like build systems and course prerequisites.
8. **Weighted Graph** — A graph where each edge carries a numerical value called weight representing cost, distance, time, or capacity. Weighted graphs are used in shortest path problems like Dijkstra's algorithm and minimum spanning tree problems like Prim's and Kruskal's.

---

4. Graph Representations

1. **Adjacency Matrix** — An n × n 2D array where adj_mat[i][j] = 1 if an edge exists between vertex i and vertex j, and 0 if no edge exists between them.
2. The adjacency matrix for an undirected graph is always symmetric because if there is an edge from vertex i to vertex j then there must also be an edge from j to i.

3. The degree of a vertex in an undirected adjacency matrix equals the sum of all values in that vertex's row.
4. **Adjacency List** — Each vertex has a linked list containing all its adjacent vertices, which makes it memory efficient especially for sparse graphs that have few edges.
5. Determining the total number of edges in a graph takes O(n + e) time using an adjacency list compared to $O(n^2)$ time with an adjacency matrix.

Feature	Adjacency Matrix	Adjacency List
Space required	Requires $O(n^2)$ space regardless of the number of edges because the full matrix is always allocated	Requires only O(n + e) space where e is the actual number of edges present in the graph
Checking if edge exists	Takes O(1) constant time because it is a direct array index lookup	Takes O(degree of vertex) time because the linked list of that vertex must be scanned
Best suited for	Dense graphs where almost all possible edges are present and the matrix is mostly filled with 1s	Sparse graphs where the number of edges is much smaller than the maximum possible n^2 edges

---

5. DFS — Depth First Search

1. DFS starts at a source vertex, marks it as visited, and then recursively visits all unvisited adjacent vertices before backtracking to explore other unvisited paths.
2. It uses a **stack** — either an explicit stack data structure or the program's implicit function call stack through recursion.
3. When the current vertex has no more unvisited neighbors, DFS backtracks to the most recently visited vertex that still has unvisited neighbors remaining.
4. DFS produces a **depth-first spanning tree** that represents the order in which vertices were first discovered during the traversal.
5. **Time Complexity**: O(n + e) with adjacency list representation and $O(n^2)$ with adjacency matrix representation.

```
Algorithm DFS(v)
{
  print v;
  visited[v] = 1;
  for (each vertex w adjacent to v)
    if (!visited[w]) DFS(w);
}
```

---

6. BFS — Breadth First Search

1. BFS starts at a source vertex and visits all of its direct neighbors first before moving on to visit the neighbors of those neighbors.
2. It uses a **queue** — the source vertex is enqueued first, then dequeued and all its unvisited neighbors are enqueued one by one.
3. BFS visits vertices **level by level** — all vertices at distance 1 from source first, then distance 2, then distance 3, and so on outward.
4. BFS always finds the **shortest path** in terms of number of edges for unweighted graphs and produces a breadth-first spanning tree.
5. **Time Complexity**: $O(n + e)$ with adjacency list representation and $O(n^2)$ with adjacency matrix representation.

```
Algorithm BFS(v)
{
  Queue q; q.insert(v); visited[v] = 1;
  while (!q.IsEmpty())
  {
    v = q.Delete();
    for (all w adjacent to v)
      if (!visited[w]) { q.insert(w); visited[w] = 1; }
  }
}
```

---

7. DFS vs BFS

Feature	DFS	BFS
Data structure used	Uses a Stack — either implicitly through recursion or as an explicit stack data structure	Uses a Queue to keep track of vertices that have been discovered but not yet fully explored

Feature	DFS	BFS
Traversal order	Goes as deep as possible along one branch before backtracking to explore other branches	Visits all vertices at the current distance level before moving to the next level outward
Shortest path	Does not guarantee the shortest path in an unweighted graph because it follows one deep branch first	Always finds the shortest path in terms of number of edges in an unweighted graph
Memory usage	Uses less memory because it only needs to store the vertices along the current path from source	Uses more memory because it stores all vertices at the current frontier level simultaneously in the queue

---

8. Minimum Spanning Tree — Overview

1. A **spanning tree** of a graph G is a subgraph that includes all n vertices of G connected with exactly n-1 edges and no cycles anywhere.
2. A **Minimum Spanning Tree (MST)** is the spanning tree whose total sum of edge weights is the minimum possible among all valid spanning trees of the graph.
3. An MST always has exactly **n-1 edges** for a graph with n vertices regardless of how many total edges the original graph contains.
4. Two main algorithms to find MST are **Kruskal's Algorithm** which is edge-based and grows the MST by adding safe edges globally, and **Prim's Algorithm** which is vertex-based and grows the MST from a starting vertex.
5. MST has real-world applications in telephone network design, electrical grid planning, road construction, and computer network routing optimization.

---

9. Kruskal's Algorithm

1. Kruskal's builds the MST by sorting all edges by weight and greedily adding the cheapest edge that does not form a cycle with the edges already selected.
2. **Step 1** — Sort all edges in the entire graph in non-decreasing order of their weight values.
3. **Step 2** — Pick the smallest remaining edge. If adding it would create a cycle in the current partial MST then discard it; otherwise add it to the MST.
4. **Step 3** — Repeat step 2 until exactly n-1 edges have been selected, which completes the minimum spanning tree.
5. **Time Complexity**: O(e log e) where e is the number of edges. Kruskal's is more efficient for **sparse graphs** that have relatively few edges compared to vertices.

10. Prim's Algorithm

1. Prim's builds the MST starting from a single vertex and repeatedly growing the tree by always adding the cheapest edge that connects a tree vertex to any non-tree vertex.
2. **Step 1** — Start with any arbitrary vertex and add it to the tree vertex set TV.
3. **Step 2** — Find the minimum cost edge that connects any vertex currently in TV to any vertex not yet in TV.
4. **Step 3** — Add that minimum cost edge and the newly reached vertex to TV. Repeat until all vertices are included in TV.
5. **Time Complexity**: $O(n^2)$ where n is the number of vertices. Prim's is more efficient for **dense graphs** that have many edges.

---

11. Kruskal's vs Prim's

Feature	Kruskal's Algorithm	Prim's Algorithm
Main approach	Edge-based approach that sorts all edges globally and picks the cheapest safe edge from the entire graph at each step	Vertex-based approach that starts from one vertex and grows the MST by always adding the cheapest edge connecting the growing tree to a new vertex
Time complexity	O(e log e) which depends primarily on the number of edges in the graph	$O(n^2)$ which depends primarily on the number of vertices in the graph
Best suited for	Sparse graphs where the number of edges is small relative to the number of vertices	Dense graphs where almost all possible edges between vertices are present
Cycle detection	Requires a union-find data structure to efficiently detect whether adding a new edge would create a cycle	Does not need explicit cycle detection because every added edge always connects to a new vertex not yet in the tree

---

12. Dijkstra's Algorithm — Single Source Shortest Path

1. Dijkstra's algorithm finds the **shortest path from one source vertex to all other vertices** in a weighted graph that has no negative edge weights anywhere.

2. It uses a **greedy approach** — at every step it picks the unvisited vertex with the minimum currently known distance from the source vertex.
3. **Initialize** — Set dist[source] = 0 and dist[all other vertices] = ∞. Mark all vertices as unvisited at the start.
4. **Update Rule** — For each newly visited vertex u, update dist[w] = min(dist[w], dist[u] + weight(u,w)) for every unvisited neighbor w of u.
5. **Time Complexity**: $O(n^2)$. Real-world applications include Google Maps navigation, network routing protocols like OSPF, and GPS shortest path systems.

```
Algorithm shortestPath(v)
{
  for (i=0; i<n; i++) { s[i]=0; dist[i]=g[v][i]; }
  s[v]=1; dist[v]=0;
  for (j=2; j<n; j++)
  {
    choose u: minimum dist[u] with s[u]==0;
    s[u]=1;
    for (each w adjacent to u with s[w]==0)
      if (dist[w] > dist[u] + g[u][w])
        dist[w] = dist[u] + g[u][w];
  }
}
```

---

13. Topological Sorting

1. Topological sort produces a linear ordering of all vertices in a **Directed Acyclic Graph (DAG)** such that for every directed edge u→v, vertex u always appears before vertex v in the ordering.
2. It is used when tasks have **precedence constraints** — meaning certain tasks must be fully completed before other tasks can begin.
3. A valid topological ordering exists **only if the graph is a DAG** — if any cycle exists in the graph then topological sort is impossible because no valid ordering can be defined.
4. **Applications**: Course prerequisite scheduling, software build dependency resolution, assembly line sequencing, and compilation order determination.
5. **Algorithm** — Repeatedly pick a vertex with in-degree 0 meaning it has no remaining predecessors, output it, remove it and all its outgoing edges from the graph, and update the in-degrees of all its neighbors.

```
for (i = 0; i < n; i++)
{
  pick vertex v with in-degree = 0;
  output v;
  delete v and all edges leaving from v;
  update in-degrees of all neighbors of v;
}
```

---

14. Symbol Table — Introduction

1. A **symbol table** is a data structure used by a compiler to store and retrieve information about all identifiers such as variable names, function names, and class names found in the source code being compiled.
2. It is defined as a collection of **name-value pairs** where the identifier name is the key and its attributes such as data type, scope, and memory address form the associated value.
3. The four core operations on a symbol table are — Insert a new key-info pair, Remove an entry by its key, Search whether a key exists, and Retrieve the full information for a given key.
4. It is used during compilation to verify that all identifiers are declared before they are used, to perform type checking, and to generate correct target machine code.
5. Identifiers are entered into the symbol table by the analysis phases of the compiler — the scanner inserts tokens, the parser adds structure information, and the semantic analyzer adds type and scope details.

---

15. Static Tree Table vs Dynamic Tree Table

1. **Static Tree Table** — All symbols and identifiers are completely known in advance before the program runs, so no insertions or deletions occur during execution. The classic example is a table of reserved keywords in a compiler.
2. **Dynamic Tree Table** — Symbols are not all known in advance and are inserted into the table as they are encountered during execution, and removed when they go out of scope or are no longer needed.
3. Static tables are typically implemented using sorted arrays combined with binary search, giving O(log n) lookup time but expensive O(n) insertion time due to array shifting.
4. Dynamic tables are implemented using self-balancing BSTs like AVL trees, giving O(log n) time for both lookup and insertion operations efficiently.

5. The choice between static and dynamic entirely depends on whether the set of identifiers is fixed and known ahead of time or whether it grows and changes dynamically during execution.

---

16. AVL Tree — Definition and Properties

1. An AVL tree named after Adelson-Velskii and Landis is a **height-balanced Binary Search Tree** where for every single node in the tree, the heights of its left and right subtrees differ by at most 1.
2. **Balance Factor (BF)** = height(left subtree) − height(right subtree). For any node to be valid in an AVL tree its balance factor must be exactly **−1, 0, or +1** — any other value means the tree is unbalanced.
3. AVL trees guarantee **O(log n)** time for search, insertion, and deletion — unlike a regular BST which can degenerate into a straight line and degrade to O(n) in the worst case.
4. Whenever an insertion causes any ancestor node's balance factor to become +2 or −2, the tree has become unbalanced and must be **rebalanced immediately using the appropriate rotation**.
5. There are exactly four types of rotations — **LL, RR, LR, and RL** — and the correct rotation is chosen based on where the newly inserted node is positioned relative to the unbalanced ancestor node.

---

17. AVL Tree — Four Rotations

1. **LL Rotation (Single Right Rotation)** — The new node is inserted in the **left subtree of the left child** of the unbalanced node A. The fix is a single right rotation at node A which brings the left child up to become the new local root of this subtree.
2. **RR Rotation (Single Left Rotation)** — The new node is inserted in the **right subtree of the right child** of the unbalanced node A. The fix is a single left rotation at node A which brings the right child up to become the new local root.
3. **LR Rotation (Double Rotation — Left then Right)** — The new node is inserted in the **right subtree of the left child** of A. The fix requires two rotations — first a left rotation on the left child to convert it to an LL case, then a right rotation on A itself.
4. **RL Rotation (Double Rotation — Right then Left)** — The new node is inserted in the **left subtree of the right child** of A. The fix requires two rotations — first a right rotation on the right child to convert it to an RR case, then a left rotation on A itself.
5. After any rotation is applied, the height of the subtree is restored to exactly what it was before the insertion, which automatically ensures the entire tree above remains balanced.

Quick Memory Trick:

- Left-Left imbalance → **LL** → Single Right Rotation at A
- Right-Right imbalance → **RR** → Single Left Rotation at A
- Left-Right imbalance → **LR** → Double Rotation (Left on child, then Right on A)
- Right-Left imbalance → **RL** → Double Rotation (Right on child, then Left on A)

SUMMARY TABLES

Asymptotic Notation Summary

Notation	Type of Bound	Mathematical Condition	What it Means
$O(g(n))$	Upper bound	$0 \leqslant f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$	f(n) grows no faster than g(n) — used to express worst case performance
$\Omega(g(n))$	Lower bound	$0 \leqslant c \cdot g(n) \leqslant f(n)$ for all $n \geqslant n_0$	f(n) grows no slower than g(n) — used to express best case performance
$\Theta(g(n))$	Tight bound	$c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$ for all $n \geqslant n_0$	f(n) grows at exactly the same rate as g(n) — the most precise bound

---

Master Theorem Cases Summary

Case	Condition	Result	Meaning
Case 1	$a < b^d$	$T(n) = \Theta(n^d)$	Work done outside the recursion dominates the total cost
Case 2	$a = b^d$	$T(n) = \Theta(n^d \log n)$	Both costs are equal in magnitude so a log factor is added
Case 3	$a > b^d$	$T(n) = \Theta(n^{\log_b(a)})$	The recursive subproblem cost dominates the total cost

---

All Algorithm Complexities

Algorithm	Best Case	Worst Case	Space
Linear Search	O(1)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(1)
Merge Sort	Θ(n log n)	Θ(n log n)	O(n)
Quick Sort	Θ(n log n)	Θ(n^2)	O(log n)
Fractional Knapsack	O(n log n)	O(n log n)	O(1)
Huffman Coding	O(n log n)	O(n log n)	O(n)
Job Sequencing	O(n log n)	O(n^2)	O(n)
Dijkstra's	O(n^2)	O(n^2)	O(n)
Prim's MST	O(n^2)	O(n^2)	O(n)
Kruskal's MST	O(e log e)	O(e log e)	O(n)
AVL Insert/Search	O(log n)	O(log n)	O(n)

End of DAA Theory Notes — All the best!