1 Asymptotic Notation

1.1 What do O(n), Ω(n), and Θ(n) mean?

1. **O(n) - Big O Notation**

- Tells us "it will never be slower than this" - upper bound on growth rate. Example: 2n + 3 is O(n) because it grows linearly

2. **Ω(n) - Big Omega Notation**

- Tells us "it will never be faster than this" - lower bound on growth rate. Example: 2n + 3 is Ω(n) because it can't grow slower than linear

3. **Θ(n) - Big Theta Notation**

- Tells us "it will always be exactly this" - tight bound. Example: 2n + 3 is Θ(n) because it's both O(n) and Ω(n)

1.2 Real Examples

- **Bubble Sort**: Best Θ(n), Worst/Avg Θ(n^2)
- **Binary Search**: Best O(1), Worst Θ(log n)
- **Linear Search**: Best O(1), Worst Θ(n)

1.3 Important Notes

- O notation can describe any bound (worst, best, or average case) - we must specify which case we're analyzing. Often it could be the worst case for an algorithm if nothing else is specified.
- We can use Θ to describe an algorithm's complexity if all its executions fall within $\Theta(f(n))$, even if theoretical best/worst cases differ
- The key is whether the actual running time is tightly bounded, not whether best/worst cases are identical

1.4 Common Time Complexities

- O(1): Constant (array access), O(log n): Logarithmic (binary search)
- O(n): Linear (linear search), O(n log n): Linearithmic (merge sort)
- O(n^2): Quadratic (bubble sort), O(2^n): Exponential (recursive Fibonacci)

1.5 Space Complexity

- Similar to time complexity but for memory usage. Example: O(n) space = memory proportional to input size

1.6 Example Questions

- **Q:** Explain what O(n), Ω(n), and Θ(n) mean.
- **A:** O(n) is an upper bound (never grows faster), Ω(n) is a lower bound (never grows slower), and Θ(n) means both bounds are tight (grows exactly at that rate).
- **Q:** Explain the difference between O(n) and Θ(n) using a concrete example.
- **A:** Consider linear search: it's O(n) because it never takes more than n steps, but it's not Θ(n) because in the best case (element found first) it takes O(1) time. For an example of Θ(n), consider a loop that always processes each element exactly once.
- **Q:** Why might an algorithm have different time complexities for its best and worst cases? Give an example.
- **A:** Algorithms often have early exit conditions or different paths based on input. For example, linear search is O(1) in best case (element found first) but Θ(n) in worst case (element not found or found last) because it might find the element immediately or need to check every position.
- **Q:** How can you determine if an algorithm's time complexity is Θ(n) rather than just O(n)?
- **A:** To prove Θ(n), you need to show both O(n) and Ω(n). This means proving the algorithm never takes more than cn steps (O(n)) AND never takes fewer than dn steps (Ω(n)) for some constants c and d. For example, a loop that always processes each element exactly once is Θ(n) because it takes exactly n steps.
- **Q:** Explain why we often focus on worst-case analysis when using O notation.
- **A:** Worst-case analysis gives us a guarantee that the algorithm will never perform worse than the bound, which is crucial for reliability and performance guarantees. It helps us prepare for the most challenging scenarios and ensures our solution will work efficiently even in the most demanding cases.

2 Stable Matching

Stable Matching is a problem in which we (for example) want to match n men and n women where each person has ranked all members of opposite sex. The goal is to find a matching where no two people would prefer each other over their current partners.

- A matching is stable if no pair (A,B) exists where:
  - A prefers B over their current match
  - B prefers A over their current match

2.1 Gale-Shapley Algorithm

- Each man proposes to his most preferred woman who hasn't rejected him
- Each woman accepts if:
  - She is unmatched, or
  - She prefers the new proposal over her current match
- Process continues until everyone is matched
- Always produces a stable matching
- Proposer-optimal, meaning the proposer gets the best possible stable matching for them.

2.2 Time Complexity

- O(n^2) where n is number of men/women
- Each man proposes at most n times
- n men making proposals
- Total: n * n = O(n^2)

2.3 Example Questions

- **Q:** Explain why the Gale-Shapley algorithm finds a stable matching.
- **A:** Gale-Shapley always finds a stable matching because it continues until everyone is matched and no pair would prefer each other over their current matches.
- **Q:** Can there be multiple stable matchings for the same set of preferences? Give an example.
- **A:** Yes, there can be multiple stable matchings. For example, with 2 men (A,B) and 2 women (X,Y), if A prefers X>Y, B prefers Y>X, X prefers B>A, and Y prefers A>B, then both (A-X, B-Y) and (A-Y, B-X) are stable matchings.
- **Q:** What happens if we reverse the roles in Gale-Shapley (women propose to men)?
- **A:** The algorithm still works but it will be women-optimal instead of men-optimal. In this algorithm the one who proposes is the one who gets the best possible stable matching.
- **Q:** Why can't a man be rejected by all women in the Gale-Shapley algorithm?
- **A:** Because there are equal numbers of men and women, and each woman can only be matched to one man. If a man was rejected by all women, it would mean all women are matched to other men, which is impossible since there are n men and n women.

3 Data Structures

3.1 Priority Queue

- A data structure that always gives the element with highest (or lowest) priority
- Internal Implementation:
  - Stored as a binary heap in a contiguous array
  - For any element at index i:
    - Left child is at index $2i + 1$
    - Right child is at index $2i + 2$
    - Parent is at index $\lfloor (i - 1)/2 \rfloor$
  - In a min-heap, each parent node must be less than or equal to its children (smallest element at root). A max-heap is the opposite - each parent is greater than or equal to its children (largest element at root)
- Main Operations:
  - **Insert(x)**:
    - Add element at the end of the array
    - "Bubble up" by swapping with parent until heap property is satisfied
    - Time complexity: O(log n)
  - **ExtractMin()** (or **ExtractMax()**):
    - Remove root element
    - Move last element to root
    - "Bubble down" by swapping with the smaller of the two children until heap property is satisfied
    - Time complexity: O(log n)
  - **Peek()**:
    - Simply return the root element (index 0)
    - Time complexity: O(1)
- Applications:
  - Dijkstra's algorithm (for shortest paths)
  - Prim's algorithm (for MSTs)
  - Task scheduling
  - Event-driven simulation

3.2 Union-Find

- A simple data structure for tracking connected components
- Implementation:
  - Parent array: index = vertex, value = parent vertex
  - Rank array: index = vertex, value = height of tree rooted at vertex
  - If a vertex is its own parent, it's a root
  - Example:
    - Parent array: [0, 0, 1, 1]
    - Rank array: [1, 2, 0, 0]
    - Means:
      - Vertices 0 and 1 are roots (they point to themselves)
      - Vertex 2's parent is 1
      - Vertex 3's parent is 1
      - Tree at 0 has height 1
      - Tree at 1 has height 2
      - So vertices 0, 1, 2, and 3 form two trees: 0 and 1,2,3
- Operations:
  - Find: Follow parent pointers until we reach a root
  - Union: Make root of smaller tree point to root of larger tree
- Optimizations:
  - Path Compression: During Find, make all nodes point directly to root
  - Union by Rank: Use rank array to always attach smaller tree to larger one
- Time Complexity: O(α(n)) where α is the inverse Ackermann function (a function that grows so slowly it's practically constant)

3.3 Hash Tables

A hash table stores key-value pairs and provides fast lookups. It uses a hash function to convert keys into array indices.

- **How it works:**
  - Hash Function: Converts key to a number
  - Modulo Operation: Converts hash to array index
  - Collision Handling: When two keys map to same index
- **Collision Handling Methods:**
  - **Separate Chaining**
    - Each array slot holds a list
    - Colliding items go in the same list
    - Simple but can get slow with long lists
  - **Open Addressing**
    - Try to find another empty slot ("probing")

    - Types of probing:
        - **Linear**: try next slot, then next, etc.
        - **Quadratic**: try slots with increasing gaps like +1, +4, +9, etc.
        - **Double hashing**: use two different hash functions - first one gives initial position, second one gives step size for probing
- **Deletion in Open Addressing:**
  - Can't just clear slot (breaks probe chain)
  - Two approaches:
    - Use tombstone (special marker)
    - Move items back (more complex)
- **Load Factor (α):**
  - α = number of items / total slots
  - Affects performance and insertion success
  - Different methods have different limits:
    - Separate chaining: works with $\alpha > 1$
    - Linear probing: works up to $\alpha \approx 0.7$-0.9
    - Quadratic probing: keep $\alpha \leq 0.5$

3.4 Example Questions

- **Q:** With open addressing, how can pairs be deleted?
- **A:** Use a special marker (tombstone) to mark deleted slots, or move elements back to maintain the probe chain.
- **Q:** What does quadratic probing mean?
- **A:** When a collision occurs, try slots at increasing squared distances (h+1, h+4, h+9, etc.) from the original hash position.
- **Q:** What does double hashing mean? Why can α be larger with double hashing than with quadratic probing?
- **A:** Use two different hash functions to determine the probe sequence; it can handle higher load factors because it provides better distribution of probes.
- **Q:** Why can't we just delete a slot in open addressing by setting it to empty? What problems would this cause?
- **A:** Setting a slot to empty would break the probe chain. If we later search for a key that was inserted after the deleted item, we would stop at the empty slot and never find the key, even though it exists in the table.
- **Q:** Compare the advantages and disadvantages of separate chaining versus open addressing.
- **A:** Separate chaining is simpler to implement and can handle any load factor, but uses more memory and can be slower due to list traversal. Open addressing is more memory efficient and can be faster due to better cache locality, but is more complex to implement and has stricter load factor limits.
- **Q:** What happens to the performance of a hash table as the load factor increases? Why?
- **A:** Performance degrades as load factor increases because there are more collisions, leading to longer probe sequences in open addressing or longer chains in separate chaining. This means more comparisons are needed to find or insert items.
- **Q:** Why might quadratic probing be better than linear probing in practice?
- **A:** Quadratic probing helps avoid primary clustering (where items cluster around the same initial positions) by spreading out the probe sequence. This leads to more even distribution of items and better performance, especially at higher load factors.

3.5 Graph Representations

- **Adjacency Matrix:**
  - 2D array where A[i][j] = 1 if edge exists between vertex i and vertex j
  - Indices i and j correspond to vertex numbers (e.g., vertex 0, 1, 2, etc.)
  - For a graph with n vertices, matrix is n × n
  - Example: If A[2][3] = 1, there's an edge from vertex 2 to vertex 3
  - Space: $\Theta(\text{Vertices}^2)$
  - Good for dense graphs (many edges)
  - Fast edge lookup: O(1)
- **Adjacency List:**
  - Array of lists, where each list contains neighbors
  - Space: $\Theta(\text{Vertices} + \text{Edges})$
  - Good for sparse graphs (few edges)
  - Slower edge lookup: O(number of edges connected to vertex)

3.6 Graph Traversal

- **Depth-First Search (DFS):**
  - Explore as far as possible along each branch before backtracking
  - Uses stack (implicitly in recursion)
  - Time: O(Vertices + Edges)
  - Applications: cycle detection, topological sort, maze solving
- **Breadth-First Search (BFS):**
  - Explore all neighbors at current depth before moving deeper
  - Uses queue
  - Time: O(Vertices + Edges)
  - Applications: shortest path (unweighted), level-order traversal

3.7 Example Questions

- **Q:** When would you choose an adjacency matrix over an adjacency list?
- **A:** Use adjacency matrix when the graph is dense (many edges) and you need fast edge lookups. The O(1) edge lookup time can be worth the extra space for dense graphs.
- **Q:** What's the main difference between DFS and BFS?
- **A:** DFS explores as far as possible along each branch before backtracking, while BFS explores all neighbors at the current depth before moving deeper. This makes BFS better for finding shortest paths in unweighted graphs.
- **Q:** Why is the time complexity of both DFS and BFS O(Vertices + Edges)?

- **A:** Both visit each vertex once and each edge once. The total work is the sum of visiting all vertices and exploring all edges.
- **Q:** In a maze-solving problem, why might DFS be better than BFS?
- **A:** DFS is better for maze-solving because it explores one path completely before backtracking, which is more memory-efficient than BFS's level-by-level approach. DFS also naturally follows the physical structure of a maze.
- **Q:** How would you detect a cycle in an undirected graph using DFS or BFS?
- **A:** Both algorithms detect cycles the same way: if you encounter a vertex that's already been visited and it's not the parent of the current vertex, then you've found a cycle. The only difference is that BFS will find the cycle through vertices at similar levels, while DFS might find it through a deeper path.
- **Q:** Why does BFS guarantee the shortest path in an unweighted graph?
- **A:** BFS explores vertices in order of their distance from the start vertex. When it first visits a vertex, it must be through the shortest path because any longer path would have to go through vertices that haven't been visited yet.

4 Shortest Paths and MSTs

4.1 Minimum Spanning Trees (MST)

- A tree that connects all vertices with minimum total edge weight
- Properties:
  - No cycles (by definition of a tree)
  - Connects all vertices
  - Has exactly Vertices-1 edges
- Applications:
  - Network design (minimizing cost of wiring/cables)
  - Road planning
  - Clustering

4.2 Creating an MST: Dijkstra's Algorithm

- Finds shortest paths from a source vertex to all other vertices
- Key idea: Always process the vertex with smallest known distance
- Process:
  - Use priority queue to track vertices by their current distance (to source (when we say distance we always mean distance to source))
  - Initialize distances array: 0 for source, ∞ for all others
  - Initialize visited array: all vertices marked as unvisited
  - For each vertex:
    - For each unvisited neighbor:
      - Calculate new distance = current vertex's distance + edge weight
      - If new distance is shorter than current best in the distance array, update distance array
      - Add neighbor to priority queue if not present, or update its distance in queue if already present (using a hash map to find its position)
    - Mark current vertex as visited in visited array
- Properties:
  - Works only with non-negative edge weights
  - Gives shortest path when all weights are positive
  - Uses a priority queue for efficiency
- Time complexity: O((Vertices + Edges)log Vertices)
- Space complexity: O(Vertices) for distances and priority queue

4.3 Jarnik's Algorithm (Prim)

- Builds MST by growing a single tree from a starting vertex
- Key idea: Always add the cheapest edge that connects to a new vertex
- Process:
  - Start with any vertex
  - Keep track of cheapest edge to each unvisited vertex
  - At each step:
    - Add vertex with cheapest connecting edge
    - Update costs to unvisited neighbors
    - Mark vertex as visited
- Properties:
  - Always maintains a single connected tree
  - Guarantees minimum total weight
  - Similar to Dijkstra's but focuses on edge weights, not path lengths
- Time complexity: O((Vertices + Edges)log Vertices)
- Space complexity: O(Vertices) for costs and priority queue

4.4 Kruskal's Algorithm

- Builds MST by adding edges in order of increasing weight
- Key idea: Always add the smallest edge that connects two different trees
- (Sorting can be reversed to get a max-spanning tree instead of min-spanning tree)
- Process:
  - Sort all edges by weight so that we can later know that we are adding edges in order of increasing weight, smallest first
  - Start with empty graph
  - For each edge in sorted order:
    - Add edge if it connects two different trees (using Union-Find to check)
    - Skip edge if it would connect vertices in the same tree
- Implementation details:
  - Use Union-Find data structure to track connected components
  - Path compression and union by rank for efficiency
- Time complexity is O(M log M) where M is number of edges and is dominated by sorting of edges.
- Properties:

  - Can handle disconnected components initially
  - Works with any graph structure
  - Produces same total weight as Prim's (though different edges)
- Real-world considerations:
  - No redundancy in final tree - if an edge fails, network becomes disconnected and needs to be recomputed
- Applications:
  - Any scenario where you need to:
    * Connect all points
    * Minimize total cost
    * Have exactly one unique path between any two points
  - For example: Network design (minimizing cost of wiring/cables), clustering, road planning

4.5 Example Questions

- **Q:** Why does Dijkstra's algorithm only work with non-negative edge weights?
- **A:** With negative weights, the "greedy" choice of always taking the shortest path might not be optimal. A longer path with negative edges could actually be shorter than a direct path with positive edges.
- **Q:** What's the main difference between Prim's and Kruskal's algorithms?
- **A:** Prim's grows a single tree by always adding the cheapest edge that connects to a new vertex, while Kruskal's builds multiple trees and merges them by adding the cheapest edge that doesn't create a cycle (aka doesn't connect two vertices in the same tree).
- **Q:** Why do we need the Union-Find data structure in Kruskal's algorithm?
- **A:** Union-Find efficiently tracks which vertices are connected, allowing us to quickly check if adding an edge would create a cycle. Without it, we'd need to do a full graph traversal to check for cycles, which would be much slower.
- **Q:** How does path compression in Union-Find improve performance?
- **A:** Path compression makes all nodes point directly to their root during a Find operation, flattening the tree structure. This makes future operations faster because we don't have to traverse long chains of parent pointers.
- **Q:** Why does Kruskal's algorithm produce an MST?
- **A:** Kruskal's algorithm always picks the smallest available edge that doesn't create a cycle. This ensures minimal total weight and guarantees all nodes are connected without cycles — which defines a Minimum Spanning Tree.
- **Q:** What is the time complexity of Kruskal's algorithm, and why?
- **A:** The time complexity is O(M log M). This is because fastest possible sorting algorithm works by repeatedly splitting the list in the middle, which takes log M steps, and at each step it processes all M edges — so the total work is $M \cdot \log M$. Sorting takes the majority of the time and the union-find operations are nearly constant time due to path compression and union by rank. This leaves us with O(M log M).
- **Q:** What happens if an included edge collapses in real applications?
- **A:** The network becomes disconnected since it's already the minimal amount of edges possible.
- **Q:** What's the main difference between Prim's and Kruskal's algorithms?
- **A:** Prim's grows a single tree by always adding the cheapest edge that connects to a new vertex, while Kruskal's builds multiple trees and merges them by adding the cheapest edge that doesn't create a cycle (aka doesn't connect two vertices in the same tree).
- **Q:** Why do we need the Union-Find data structure in Kruskal's algorithm?
- **A:** Union-Find efficiently tracks which vertices are connected, allowing us to quickly check if adding an edge would create a cycle. Without it, we'd need to do a full graph traversal to check for cycles, which would be much slower.
- **Q:** What happens if the graph is disconnected? Can Kruskal's still find an MST?
- **A:** The algorithm will naturally handle disconnected components by not adding edges between them creating a collection of MSTs
- **Q:** Can Dijkstra's algorithm handle negative edge weights?
- **A:** No, Dijkstra's algorithm cannot handle negative edge weights. This is because the algorithm assumes that the shortest path to a vertex is found when we first visit it. With negative weights, a longer path with negative edges could actually be shorter than a direct path with positive edges, breaking this assumption.
- **Q:** Can both Kruskal's and Prim's algorithms handle negative edge weights?
- **A:** Yes. Because they only care about the relative ordering of edge weights when choosing which edges to add to the tree, not their absolute values. The minimum spanning tree will still be found correctly regardless of whether the weights are positive or negative.