EDAF05 Algorithms, Data Structures and Time & Space Complexity

Adam Tovatt 2025-06-14

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 · 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.

5 Divide and Conquer & Convex Hull

5.1 Divide and Conquer

- Key steps:
  - Divide: Split problem into smaller instances
  - Conquer: Solve subproblems recursively
  - Combine: Merge solutions into final answer
- Common examples include merge sort, quick sort, binary search, and finding closest pair of points
- Master Theorem:
  - When we solve a problem by breaking it into smaller pieces, we often get a pattern like:
    * To solve a problem of size n, we need to solve some number of subproblems (let's call this number a)
    * Each of these subproblems is of size n/b (where

    b is how much we divide the problem size by)
    * Plus we need to do some extra work $f(n)$ to combine the results
  - This pattern is called a recurrence relation
  - Example: In merge sort:
    * We split array in half (so $b = 2$, because we divide size by 2)
    * We solve two subproblems (so $a = 2$):
      · Sort the left half of the array
      · Sort the right half of the array
    * We merge the sorted halves (which takes n time, so $f(n) = n$)
    * So $T(n) = 2T(n/2) + n$
  - Different example: In binary search:
    * We split array in half (so $b = 2$)
    * We solve only one subproblem (so $a = 1$):
      · Search in either the left half OR the right half
      · We don't need to search both halves because we can determine which half to look in
    * We do constant work to compare and choose which half (so $f(n) = 1$)
    * So $T(n) = T(n/2) + 1$
  - The Master Theorem formula:
    * If $T(n) = aT(n/b) + f(n)$ where:
      · $a \geq 1$ (we must have at least one subproblem)
      · $b > 1$ (we must divide the problem into smaller pieces)
      · $f(n)$ is asymptotically positive ($f(n)$ is positive for most values of n, especially the large ones we care about)
    * Then $T(n)$ is one of these three cases:
      · If the extra work $f(n)$ is much smaller than $n^{\log_b(a)}$, then $T(n) = \Theta(n^{\log_b(a)})$
      · If the extra work $f(n)$ is about the same size as $n^{\log_b(a)}$, then $T(n) = \Theta(n^{\log_b(a)} \log n)$
      · If the extra work $f(n)$ is much larger than $n^{\log_b(a)}$, then $T(n) = \Theta(f(n))$

5.2 Convex Hull

A convex hull is the smallest convex polygon that contains all points in a set. It has several important properties: all points must lie on or inside the hull, no interior angles can be greater than 180 degrees, and it has the minimum possible perimeter that can enclose all points.

5.2.1 Jarvis March (Gift Wrapping)

- Take the leftmost point as starting point and imagine a horizontal line to the right from it
- For each other point, draw a line from the starting point to this other point
- Measure the angle between the horizontal line and this new line
- Take the other point that creates the largest counterclockwise angle with the horizontal line and consider it part of the hull, the edge to it is the new "current edge"
- For each subsequent point on the hull:
  - Find the point that makes the largest counterclockwise angle with the current edge
  - This is done by comparing angles between the current edge and all other points
  - The point with the largest angle will be the next point on the hull and the edge to it the new "current edge"
- Continue until we return to the starting point
- Time: $O(nh)$ where n is the total number of points and h is number of points on the hull
- Worst case: $O(n^2)$ when all points are on the hull (i.e., when $h = n$)
- Optimization: We don't actually need to calculate angles. Instead, we can use cross products to determine which point makes the largest counterclockwise turn. For points A, B, C: if cross product $(B - A) \times (C - A)$ is positive, C is counterclockwise from B and so on.

5.2.2 Graham Scan

A polar angle is the angle between a point and a reference point relative to a reference direction (usually horizontal right).

Steps:

- Find the point with lowest y-coordinate (if tie, take leftmost) - this is our reference point. This point is guaranteed to be on the hull.
- Sort all other points by their polar angle in ascending order (counterclockwise) from the reference point (use cross product for speed). This means we'll process points starting from the right of the reference point and moving counterclockwise.
- If two points have same polar angle, keep only the furthest one from the reference point (the closer point will be inside the hull)
- Start with the first three points in a stack
- For each remaining point:
  - While the last three points in stack make a right turn (negative cross product) or are collinear (zero cross product), pop the middle point
  - Push the current point onto stack
- The stack now contains the convex hull points in counterclockwise order

The time complexity is $O(n \log n)$ due to sorting, and space complexity is $O(n)$ for the stack. Graham Scan has several advantages over Jarvis March: it's more efficient when many points are on the hull, has a guaranteed $O(n \log n)$ time complexity, and is relatively simple to implement.

5.2.3 Preparata-Hong (Merge Hull)

Preparata-Hong is a divide-and-conquer algorithm for finding the convex hull that works in both 2D and 3D. It's particularly notable for being one of the first efficient algorithms for 3D convex hulls. The process involves splitting the points into two halves, finding the hulls recursively, and then merging the hulls by finding "bridge" edges. The time complexity is $O(n \log n)$. Its applications include 3D modeling, computer graphics, and collision detection. The algorithm is well-suited for parallelization because the recursive subproblems can be solved independently in parallel, and the merging step can also be parallelized.

Process for 2D (3D is similar but more complex):

- Split points into two halves (for example by picking a point and dividing the remaining points into those with smaller and larger x-coordinates)
- Find hulls recursively:
  - Base case: if we have 3 or fewer points, order them clockwise using cross products (like in Graham Scan). This is already a convex hull.
  - Otherwise, split the points and recursively find the hulls of each half
- Merge hulls by finding “bridge” edges:
  - Find the upper bridge: start with rightmost point of left hull (highest x-coordinate) and leftmost point of right hull (lowest x-coordinate)
  - Look at next points clockwise on left hull and counterclockwise on right hull
  - For each hull, if the line between current points would go below its next point, move to that next point
  - Repeat until we find points where the line between them would go above their next neighbors
  - Find the lower bridge: start with the same start points as before
  - Look at next points counterclockwise on left hull and clockwise on right hull
  - For each hull, if the line between current points would go above its next point, move to that next point
  - Repeat until we find points where the line between them would go below their next neighbors
  - Remove all points between the bridges that are not part of the new hull, maintaining clockwise order

5.3 Finding Nearest Points

- Problem: Find the smallest distance between any two points in a 2D plane
- Divide-and-conquer approach:
  - Sort points by x and y coordinates ($O(n \log n)$)
  - Split points into left and right halves
  - Find closest pairs in each half recursively
  - Check for closer pairs that cross the dividing line
- Time complexity: $O(n \log n)$
  - $a = 2$ (we split into two subproblems)
  - $b = 2$ (each subproblem is half the size)
  - $f(n) = n$ (we need to check points in the strip)
  - Using Master Theorem: $n^{\log_2(2)} = n$, and $f(n) = n$, so we’re in case 2
  - Therefore $T(n) = \Theta(n \log n)$
- Key insight: In the strip around the dividing line, each point only needs to be compared with at most 7 other points
- Base case: When we have 3 or fewer points, use brute force

5.4 Example Questions

- **Q:** Explain what is meant by a divide-and-conquer algorithm.
- **A:** Divide the problem into smaller subproblems, solve them recursively, and combine their solutions to solve the original problem.
- **Q:** Explain what the Master theorem is about.
- **A:** The Master theorem provides a way to solve recurrence relations of the form $T(n) = aT(n/b) + f(n)$ by comparing the growth of $f(n)$ with $n^{\log_b(a)}$.
- **Q:** What is a convex hull?
- **A:** The smallest convex polygon containing all points, with no interior angles greater than 180 degrees.
- **Q:** Explain the Graham scan algorithm.
- **A:** Graham scan sorts points by polar angle, then uses a stack to build the hull by removing points that create concave angles.
- **Q:** Explain the main ideas of the Preparata-Hong algorithm.
- **A:** Preparata-Hong is a divide-and-conquer algorithm for 3D convex hull that splits points, finds hulls recursively, and merges them efficiently.
- **Q:** Why is it important to compare either α or β with γ first in different situations? What is likely to happen otherwise?
- **A:** When doing geometric computations, especially with floating-point numbers, the order of comparisons can affect numerical stability. For example, when comparing three angles α, β, and γ, we should always compare them in a consistent order (like always comparing α with γ first, then β with γ). This helps avoid floating-point errors that can occur due to different orders of operations. If we don’t maintain a consistent order, we might get different results for the same geometric situation due to rounding errors in floating-point arithmetic.
- **Q:** How can you know if a point p is between q and r on a line?
- **A:** Use cross product to check collinearity and dot product to check if p is between q and r on the line.
- **Q:** How can you know the direction (left, right, or straight) when going from a point pr through ps to pt?
- **A:** Use the cross product of vectors ps-pr and pt-ps; positive means left turn, negative means right turn, zero means straight.
- **Q:** What’s the difference between Jarvis March and Graham Scan in terms of when each is more efficient?
- **A:** Jarvis March is more efficient when the number of points on the hull (h) is small, as it runs in O(nh) time. Graham Scan is better when most points are on the hull, as it runs in O(n log n) time regardless of hull size. Graham Scan is generally preferred for large datasets as its worst case is better.
- **Q:** How does the time complexity of finding the closest pair of points using divide and conquer compare to a brute force approach?

- **A:** The divide and conquer approach runs in O(n log n) time, while brute force would be O(n^2). The improvement comes from only needing to check a limited number of points in the strip between divided regions.
- **Q:** Why do we need to check points in the 'strip' when finding the closest pair of points? What's the maximum number of points we need to check in the strip?
- **A:** We need to check the strip because the closest pair might be split across the dividing line. The maximum number of points we need to check in the strip is 7, as any more points would violate the minimum distance property we've already established.
- **Q:** In the Master Theorem, what happens when $f(n)$ grows faster than $n^{\log_b(a)}$? Give an example.
- **A:** When $f(n)$ grows faster than $n^{\log_b(a)}$, the time complexity is dominated by $f(n)$, giving us $T(n) = \Theta(f(n))$. For example, if $T(n) = 2T(n/2) + n^2$, then $n^{\log_2(2)} = n$, and since n^2 grows faster than n, we get $T(n) = \Theta(n^2)$.
- **Q:** How does the choice of starting point affect Jarvis March? Does it matter which point we start with?
- **A:** The choice of starting point doesn't affect the correctness of Jarvis March, but it can affect the number of points we need to check. Starting with the leftmost point is common because it's guaranteed to be on the hull, but any point on the hull would work.
- **Q:** How can we handle collinear points in convex hull algorithms? What special cases do we need to consider?
- **A:** For collinear points, we typically want to keep only the outermost points on the hull. This means when three points are collinear, we should keep the two endpoints and remove the middle point. We need to handle this in both Jarvis March and Graham Scan to avoid including unnecessary points.
- **Q:** Why is it important that the Master Theorem's f(n) is asymptotically positive? What could go wrong if it wasn't?
- **A:** The Master Theorem requires f(n) to be asymptotically positive because negative work doesn't make sense in the context of algorithm complexity. If f(n) could be negative, it would mean the algorithm is doing "negative work" in some steps, which doesn't correspond to any real computation. This would make the recurrence relation meaningless for analyzing algorithm complexity.

6 Dynamic Programming

6.1 Main Ideas

Dynamic Programming is a way to solve problems by breaking them into smaller subproblems and storing solutions to avoid solving them again. Key characteristics:

- Like divide-and-conquer, but subproblems overlap
- Store solutions to subproblems to avoid solving them again
- Use when:
  - Problem has overlapping subproblems
  - Subproblems can be combined to solve main problem
  - We can write a recurrence relation

6.2 Recurrence Relations

A recurrence relation is like a recipe that tells you how to solve a problem using solutions to smaller versions of the same problem. It has two parts:

- The formula that shows how to combine smaller solutions
- The base cases (smallest problems that we can solve directly)

6.2.1 Examples

Fibonacci

- Formula: $fib(n) = fib(n-1) + fib(n-2)$
- Base cases: $fib(0) = 0$, $fib(1) = 1$

Longest Increasing Subsequence

- Formula: $lis(i) = 1 + \max(lis(j))$ where $j < i$ and $array[j] < array[i]$
- Base case: $lis(0) = 1$

Edit Distance

- Formula: $ed(i, j) = \min(\text{delete}, \text{insert}, \text{replace})$
- Where:
  - delete $= ed(i-1, j) + 1$
  - insert $= ed(i, j-1) + 1$
  - replace $= ed(i-1, j-1) + cost$
- Base case: $ed(0, 0) = 0$

6.3 Two Approaches

6.3.1 Top-down (memoization):

- Start with main problem
- Solve subproblems as needed
- Store results in a table

6.3.2 Bottom-up (tabulation):

- Start with smallest subproblems
- Build up to main problem
- Fill table systematically

6.4 Steps to solve a DP problem:

1. Identify the subproblems
2. Write the recurrence relation
3. Decide on approach (top-down or bottom-up)
4. Implement the solution
5. Analyze time and space complexity

6.5 Sequence Alignment

Sequence alignment is finding the best way to align two strings by inserting gaps.

- Used to compare DNA sequences, protein sequences, or text
- Goal: Maximize similarity or minimize cost of alignment

6.5.1 Process:

- Use 2D table where cell (i,j) represents best alignment of first i characters of string 1 and first j characters of string 2
- For each cell, consider three options:
  - Match/mismatch: align current characters
  - Gap in first string: skip character in string 1
  - Gap in second string: skip character in string 2

6.5.2 Time complexity:

$O(N \times M)$ where N and M are string lengths

6.5.3 Applications:

- Bioinformatics: comparing DNA or protein sequences
- Text processing: finding similar words or documents
- Version control: finding differences between files

6.6 Example Questions

- **Q:** Explain what is meant by dynamic programming.
- **A:** Solving a problem by breaking it into overlapping subproblems and storing their solutions to avoid redundant calculations.
- **Q:** What is sequence alignment and how can it be done?
- **A:** Sequence alignment finds the best way to align two strings by inserting gaps; it's done using dynamic programming to maximize similarity or minimize cost.
- **Q:** What are the key characteristics of a problem that makes it suitable for dynamic programming?
- **A:** A problem is suitable for DP if it has overlapping subproblems (same subproblems are solved multiple times) and optimal substructure (optimal solution can be constructed from optimal solutions of subproblems).
- **Q:** Compare and contrast top-down (memoization) and bottom-up (tabulation) approaches in dynamic programming.
- **A:** Top-down starts with the main problem and solves subproblems as needed, storing results in a table. Bottom-up starts with smallest subproblems and builds up to the main problem, filling the table systematically. Top-down can be more intuitive and only computes needed subproblems, while bottom-up avoids recursion overhead and can be more space-efficient.
- **Q:** In the Edit Distance problem, explain why we need to consider all three operations (delete, insert, replace) and how they contribute to the final solution.
- **A:** We need all three operations because they represent different ways to transform one string into another. Delete removes a character from the first string, insert adds a character to the first string, and replace changes a character in the first string. The algorithm chooses the minimum cost among these operations at each step to find the optimal transformation sequence.
- **Q:** How does the time and space complexity of a dynamic programming solution typically compare to a naive recursive solution?
- **A:** A naive recursive solution often has exponential time complexity due to solving the same subproblems multiple times. Dynamic programming reduces this to polynomial time by storing solutions to subproblems. Space complexity is typically O(n) or O(n^2) for DP, while naive recursion can use O(n) space for the call stack plus additional space for redundant computations.
- **Q:** Explain how you would modify the Edit Distance algorithm to also return the sequence of operations that leads to the minimum distance.
- **A:** We can add a second table to store the operation (delete, insert, or replace) that led to each cell's value. When filling the table, we record which operation gave the minimum value. After computing the minimum distance, we can trace back through the table following these recorded operations to reconstruct the sequence.
- **Q:** What is the difference between overlapping subproblems and independent subproblems? Why is this distinction important for dynamic programming?
- **A:** Overlapping subproblems occur when the same subproblem is solved multiple times in a recursive solution. Independent subproblems are solved only once. This distinction is crucial because dynamic programming is only beneficial for problems with overlapping subproblems - there's no need to store solutions if each subproblem is solved exactly once.

7 Network Flow

7.1 Main Ideas

Network flow is about finding the maximum amount of flow that can be sent from a source node to a sink node in a directed graph where each edge has a capacity.

7.1.1 Applications

- Finding maximum matching in bipartite graphs
- Finding maximum number of edge-disjoint paths
- Modeling traffic flow in networks
- Finding minimum cut in a graph
- Modeling water pipes and electrical systems
- Ecological applications (nutrient flow)

7.2 Ford-Fulkerson Algorithm

7.2.1 Main Idea

Keep finding paths with available capacity and add flow until no more paths exist.

7.2.2 Steps

1. Start with zero flow
2. Find a path from source to sink with available capacity
3. Add flow along this path
4. Update residual capacities
5. Repeat until no more paths exist

7.2.3 Residual Graph

- Forward edges: remaining capacity
- Backward edges: current flow (can be "undone")

7.2.4 Time Complexity

$O(E \times \text{max\_flow})$ where:

- E is number of edges
- max_flow is the maximum possible flow
- Each path finding takes $O(E)$ time
- We might need to find max_flow paths

7.2.5 Correctness

The Ford-Fulkerson algorithm is correct because:

- It always terminates (flow can only increase, and there's a maximum possible flow)
- When it terminates, no augmenting path exists
- By the max-flow min-cut theorem, this means we've found the maximum flow
- The residual graph allows us to "undo" flow if we find a better path

7.3 Bipartite Graph Matching

A bipartite graph is a graph where we can split all the vertices into two separate groups (let's call them group L and group R), and every edge in the graph must connect a vertex from one group to a vertex in the other group. In other words, you can't have edges between vertices in the same group. This structure makes bipartite graphs perfect for modeling matching problems where we need to pair things from one group with things from another group, like matching students to courses or workers to jobs.

7.3.1 Problem Definition

Given a bipartite graph $G = (L \cup R, E)$, where L and R are our two groups of vertices and E is the set of all edges connecting vertices between these groups, find the maximum number of edges that can be matched such that no two edges share a vertex.

7.3.2 Reduction to Max Flow

1. Add source node connected to all vertices in L
2. Add sink node connected from all vertices in R
3. Set all edge capacities to 1
4. Find maximum flow
5. The edges with flow 1 form the maximum matching

7.3.3 Time Complexity

$O(VE)$ where:

- V is number of vertices
- E is number of edges
- Each augmenting path increases flow by 1
- Maximum flow is at most $V/2$

7.3.4 Applications

- Job assignment problems
- Dating/matching applications
- Resource allocation
- Task scheduling

7.4 Goldberg-Tarjan (Push-Relabel)

7.4.1 Key Components

- Height labels for nodes
- Excess flow at each node
- Push flow to lower neighbors
- Relabel nodes when stuck

7.4.2 Time Complexity

$O(V^2 \times E)$ where:

- V is number of vertices
- E is number of edges
- Better than Ford-Fulkerson for large capacities

7.4.3 Advantages

- Pushes full available flow in each step
- Not affected by large edge capacities
- Often faster in practice

7.5 Handling Large Capacities

7.5.1 Why Ford-Fulkerson Can Be Slow

- Basic version increments flow by 1 unit at a time
- With large capacities (e.g., 1,000,000), needs many iterations
- Example: If max flow is 1,000,000, might need 1,000,000 iterations

7.5.2 Solutions

- Capacity scaling:
  - Begin with largest power of 2 less than max capacity
  - Halve the step size each iteration
  - Reduces iterations from $O(\text{max\_flow})$ to $O(\log \text{max\_flow})$
- Use Push-Relabel algorithm instead
- Use binary search over possible flow values

7.5.3 When to Use Which Algorithm

- Ford-Fulkerson with capacity scaling:
  - Good for sparse graphs
  - Easy to implement
  - Works well with small to medium capacities
- Push-Relabel:
  - Better for dense graphs
  - Better for very large capacities
  - More complex to implement
  - Often faster in practice

7.6 Example Questions

- **Q:** What is network flow about? Give an example of when it can be used.
- **A:** Network flow finds the maximum flow from source to sink in a directed graph with edge capacities. It's used for maximum matching, minimum cut, and traffic optimization.
- **Q:** Explain the Ford-Fulkerson algorithm and why it

is correct. What is its time complexity, and why?

- **A:** Ford-Fulkerson repeatedly finds augmenting paths and adds flow until no more paths exist. It's correct because it always finds the maximum flow. The time complexity is $O(E \times \text{max\_flow})$ because each path finding takes $O(E)$ time and we might need to find max_flow paths.
- **Q:** Explain the Goldberg-Tarjan (preflow-push) algorithm and why it is correct.
- **A:** Goldberg-Tarjan uses node heights and excess flow, pushing flow to lower neighbors and relabeling when stuck. It's correct because it maintains the height property and terminates with maximum flow.
- **Q:** What is a bipartite graph, and how can you determine if a graph is bipartite?
- **A:** A bipartite graph can be divided into two sets with no edges within the same set. You can determine if a graph is bipartite using BFS or DFS with two colors, coloring each vertex opposite to its neighbors. If you can't do this coloring without conflicts, the graph isn't bipartite.
- **Q:** What is the relationship between maximum flow and minimum cut in a network?
- **A:** The maximum flow equals the capacity of the minimum cut (max-flow min-cut theorem). This means the bottleneck in the network is determined by the minimum cut.
- **Q:** Why do we need backward edges in the residual graph for Ford-Fulkerson?
- **A:** Backward edges allow us to "undo" flow if we find a better path. They represent the current flow that can be redirected, which is crucial for finding the optimal solution.
- **Q:** In bipartite matching, why do we set all edge capacities to 1 when reducing to max flow?
- **A:** Setting capacities to 1 ensures that each vertex can only be matched once, as the flow through any edge can be at most 1. This directly corresponds to the matching constraint where no two edges can share a vertex.
- **Q:** What is the difference between a maximum matching and a perfect matching in a bipartite graph?
- **A:** A maximum matching is the largest possible set of edges where no two edges share a vertex. A perfect matching is a maximum matching where every vertex is matched (only possible if both groups have the same size).