Exam 2 Study Guide - Graphs, Greedy and Backtracking

Note: You will be provided with pseudocode from class for these many of these algorithms (karatsuba, dijkstras, heap-shiftdown, n-queens) on the last page of the exam!

Graph Fundamentals

Graph Representation

Adjacency List

- **Structure**: Array of lists, one per vertex
- **Space Complexity**: O(V + E)
- **Edge lookup**: O(degree(v))
- **Best for**: Sparse graphs (E << V^2)

Adjacency Matrix

- **Structure**: V×V matrix, entry (i,j) = 1 if edge exists
- **Space Complexity**: O(V^2)
- **Edge lookup**: O(1)
- **Best for**: Dense graphs, frequent edge lookups

Graph Types

- **Undirected**: Edges have no direction (symmetric adjacency matrix)
- **Directed**: Edges have direction (arrows)
- **Weighted**: Edges have associated costs/weights
- **Unweighted**: All edges treated equally (or weight = 1)

Graph Properties

- **Path**: Sequence of vertices connected by edges
- **Cycle**: Path that starts and ends at same vertex
- **Connected**: Path exists between any two vertices (undirected)
- **Strongly Connected**: Path exists in both directions between any two vertices (directed)
- **Tree**: Connected graph with no cycles (E = V - 1)

Breadth-First Search (BFS)

Algorithm Overview

Purpose: Explore graph level-by-level from a starting vertex

Key Characteristics:

- Uses a **queue** (FIFO)

- Visits vertices in order of increasing distance from start
- Finds **shortest path** in unweighted graphs
- Non-recursive (iterative)

BFS Algorithm Steps

1. Initialize all vertices as unvisited
2. Mark start vertex as visited, add to queue
3. While queue not empty:
    - Dequeue vertex v
    - Process v (record visit order)
    - For each unvisited neighbor w of v:
        - Mark w as visited
        - Add w to queue
        - Set predecessor[w] = v

Running Time

- **Time Complexity**: O(V + E)
    - Each vertex visited once: O(V)
    - Each edge examined once: O(E)
- **Space Complexity**: O(V) for queue and visited array

BFS for Shortest Paths

- In **unweighted graphs**, BFS finds shortest path
- Distance from start to vertex v = level at which v is discovered
- Reconstruct path using predecessor array

Depth-First Search (DFS)

Algorithm Overview

Purpose: Explore graph by going as deep as possible before backtracking

Key Characteristics:

- Uses a **stack** (LIFO) - often implemented via recursion
- Explores one branch completely before trying another
- Useful for cycle detection, topological sorting, strongly connected components
- Can be recursive or iterative

DFS Algorithm Steps (Recursive)

```
DFS(vertex v):
    mark v as visited
    process v (record visit order)
    for each neighbor w of v:
```

```
        if w is not visited:
            DFS(w)
```

DFS Algorithm Steps (Iterative)

1. Initialize all vertices as unvisited
2. Push start vertex onto stack
3. While stack not empty:
   - Pop vertex v
   - If v not visited:
     - Mark v as visited
     - Process v (record visit order)
     - Push all unvisited neighbors of v onto stack

Running Time

- **Time Complexity**: O(V + E)
  - Each vertex visited once: O(V)
  - Each edge examined once (or twice for undirected): O(E)
- **Space Complexity**: O(V) for recursion stack/explicit stack

BFS vs DFS Comparison

Feature	BFS	DFS
Data Structure	Queue	Stack (or recursion)
Exploration	Level-by-level	Deep then backtrack
Shortest Path	Yes (unweighted)	No
Memory Usage	Higher (stores level)	Lower (path only)
Use Cases	Shortest path, level-order	Cycle detection, topological sort

Dijkstra's Algorithm

Algorithm Overview

Purpose: Find shortest paths from start vertex to all other vertices in **weighted graph with non-negative weights**

Key Characteristics:

- **Greedy algorithm** - always picks closest unvisited vertex
- Uses **priority queue (min-heap)** to select next vertex
- Maintains **key values** (current shortest distance) for each vertex
- Maintains **predecessor** array to reconstruct paths
- **Does not work with negative edge weights**

Dijkstra's Algorithm Steps

1. **Initialize**:
    - Set key[start] = 0, all other keys = ∞
    - Set all predecessors to null
    - Add all vertices to min-heap (priority queue)
2. **Main Loop** (while heap not empty):
    - Extract vertex u with minimum key value
    - For each neighbor v of u:
        - Calculate new_distance = key[u] + weight(u, v)
        - If new_distance < key[v]:
            - Update key[v] = new_distance
            - Update predecessor[v] = u
            - Decrease key of v in heap
3. **Result**:
    - key[v] = shortest distance from start to v
    - Reconstruct path by following predecessors backwards

Running Time

- **Time Complexity**: O((V + E) log V) with binary heap
    - Extract-min: O(log V) × V times = O(V log V)
    - Decrease-key: O(log V) × at most E times = O(E log V)
    - Total: O((V + E) log V)

Key Insights

- **Greedy Property**: Once a vertex is removed from heap, its shortest path is finalized
- **Optimal Substructure**: All sub-paths of a shortest path are also shortest paths

Dijkstra's Algorithm Design Technique

Greedy Algorithm:

- Makes locally optimal choice at each step
- Selects vertex with minimum key value
- Never reconsiders once a vertex is processed
- Greedy choice: "Visit closest unvisited vertex next"

Greedy Algorithms

Greedy Algorithm Characteristics

Core Principle: Make the locally optimal choice at each step, hoping to find a global optimum

Key Properties:

- **Greedy Choice Property**: A global optimum can be reached by making locally optimal choices

- **Optimal Substructure**: An optimal solution contains optimal solutions to subproblems
- **Never backtracks**: Once a choice is made, it's never reconsidered
- **Efficiency**: Often runs in polynomial time

Proving Correctness:

1. **Greedy Choice Property**: Show that making the greedy choice leaves a subproblem of the same form
2. **Optimal Substructure**: Prove that combining the greedy choice with an optimal solution to the subproblem yields an optimal solution to the original problem

Interval Scheduling Problem

Problem: Given n intervals with start and finish times, select the maximum number of non-overlapping intervals.

Input: Set of intervals $\{(s_1, f_1), (s_2, f_2), ..., (s_n, f_n)\}$ where s_i = start time, f_i = finish time

Goal: Find maximum-size subset of mutually compatible (non-overlapping) intervals

Greedy Strategy: Always select the interval with the **earliest finish time** among remaining compatible intervals

Algorithm:

1. Sort intervals by finish time ($f_1 \le f_2 \le ... \le f_n$)
2. Initialize result set S = {interval 1}
3. For each interval i from 2 to n:
    - If interval i is compatible with all intervals in S ($s_i \ge$ finish time of last interval in S):
        - Add interval i to S
4. Return S

Running Time: O(n log n) for sorting + O(n) for selection = **O(n log n)**

Why This Works:

- Selecting earliest finish time leaves maximum room for future intervals
- Greedy choice is always part of some optimal solution
- Can prove by exchange argument: any optimal solution can be transformed to include the greedy choice

Minimum Cost to Connect Sticks

Problem: Given n sticks of various lengths, connect them all into one stick. Cost to connect two sticks = sum of their lengths. Find minimum total cost.

Input: Array of stick lengths $[s_1, s_2, ..., s_n]$

Goal: Minimize total cost of connecting all sticks

Greedy Strategy: Always connect the two **shortest sticks** available

Algorithm:

1. Create a min-heap from all stick lengths
2. Initialize total_cost = 0
3. While heap has more than one stick:
   - Extract two smallest sticks: stick1 and stick2
   - cost = stick1 + stick2
   - Add cost to total_cost
   - Insert combined stick (length = cost) back into heap
4. Return total_cost

Running Time:

- Build heap: O(n)
- n-1 iterations, each with 2 extract-min + 1 insert: O(n log n)
- **Total: O(n log n)**

Example:

```
Sticks: [2, 4, 3]

Step 1: Connect 2 and 3 → cost = 5, sticks = [5, 4], total = 5
Step 2: Connect 5 and 4 → cost = 9, sticks = [9], total = 14

Total cost: 14
```

Why This Works:

- Sticks connected earlier are counted multiple times in total cost
- Minimizing early connections (by using shortest sticks) minimizes total cost
- Similar to Huffman coding tree construction
- Can prove optimal by induction on number of sticks

Backtracking Algorithms

N-Queens Problem

Problem: Place n queens on an n×n chessboard such that no two queens attack each other.

Constraints: Queens can attack any piece in the same:

- Row
- Column
- Diagonal (both directions)

Goal: Find a valid placement of all n queens (or determine if one exists in a given subtree)

Backtracking Strategy:

- Place queens one column at a time
- For each column, try each row position
- If a position is safe, place queen and recurse to next column

- If no safe position exists in current column, backtrack to previous column
- Prune branches early when a placement violates constraints

Note: Exam questions will specify whether to use 0-based or 1-based indexing. Examples below use 1-based indexing.

N-Queens Algorithm Steps

```
NQueens(col, board):
    if col > n:
        return true  // All queens placed successfully

    for row from 1 to n:
        if isSafe(row, col, board):
            board[col] = row  // Place queen
            if NQueens(col + 1, board):
                return true
            // Backtrack: remove queen (implicit when trying next row)

    return false  // No valid placement found

isSafe(row, col, board):
    for prevCol from 1 to col-1:
        prevRow = board[prevCol]
        // Check same row
        if prevRow == row:
            return false
        // Check diagonals
        if abs(prevRow - row) == abs(prevCol - col):
            return false
    return true
```

Tracing N-Queens Search Tree

Key Concepts for Exam:

1. **Board Representation**: Array where board[i] = j means "queen in column i is at row j"
2. **Search Tree Structure**:
   - Each level represents a column (1 through n)
   - Each branch represents a row choice (1 through n)
   - Leaves represent complete or failed placements
3. **Determining Valid Solutions in Subtree**:
   - Start from given partial board state
   - Check if current state is valid (no conflicts)
   - If invalid, subtree has NO solutions
   - If valid, trace remaining columns systematically

Example Trace for 4-Queens (1-indexed):

```
Col 1: Try rows 1, 2, 3, 4
  |
  ├─ Q at (row=1, col=1)
  |    Col 2: Try rows 1, 2, 3, 4
  |     |
  |     ├─ row=1? NO - same row as (1,1)
  |     ├─ row=2? NO - diagonal attack with (1,1)
  |     ├─ row=3? YES - safe
  |     |    Col 3: Try rows 1, 2, 3, 4
  |     |     |
  |     |     ├─ row=1? NO - same row as (1,1)
  |     |     ├─ row=2? NO - diagonal with (3,2)
  |     |     ├─ row=3? NO - same row as (3,2)
  |     |     ├─ row=4? NO - diagonal with (3,2)
  |     |     └─ BACKTRACK - no valid row in col 3
  |     |
  |     └─ row=4? YES - safe
  |          Col 3: ...
  |          (Continue exploring...)
  |
  ├─ Q at (row=2, col=1)
  |    Col 2: ...
  |    (Contains solution: board = [2,4,1,3])
  ...
```

Checking for Valid Solutions in a Subtree

Step-by-Step Process:

1. **Verify Current State**:
   - Check all placed queens for conflicts
   - If conflict exists, answer is NO
2. **Identify Remaining Columns**:
   - Count how many columns still need queens
   - These form the subtree to explore
3. **For Each Remaining Column**:
   - Try each row systematically (1 through n)
   - Check if position is safe against ALL previously placed queens
   - If safe, recursively check next column
   - If unsafe, skip this branch (pruning)
4. **Termination Conditions**:
   - **Success**: All columns filled with no conflicts

   - **Failure**: Current column has no safe rows (backtrack)

Example Problem (1-indexed): Given board state [2, 4, ?, ?] for 4-Queens (columns 1-2 filled, columns 3-4 empty), does subtree contain valid solution?

```
Current state (board = [2, 4, ?, ?]):
. . ? ?   (row 1)
Q . ? ?   (row 2, col 1)
. . ? ?   (row 3)
. Q ? ?   (row 4, col 2)

Col 3 options:
- Row 1: Safe? Check against (2,1) and (4,2)
  - Not same row as 2 or 4 ✓
  - Diagonal from (2,1)? |2-1| == |1-3|? 1 ≠ 2 ✓
  - Diagonal from (4,2)? |4-1| == |2-3|? 3 ≠ 1 ✗ NO - diagonal conflict

- Row 2: Safe? Check against (2,1) and (4,2)
  - Same row as (2,1)? YES ✗ NO - row conflict

- Row 3: Safe? Check against (2,1) and (4,2)
  - Not same row as 2 or 4 ✓
  - Diagonal from (2,1)? |2-3| == |1-3|? 1 ≠ 2 ✓
  - Diagonal from (4,2)? |4-3| == |2-3|? 1 == 1 ✗ NO - diagonal conflict

- Row 4: Safe? Check against (2,1) and (4,2)
  - Same row as (4,2)? YES ✗ NO - row conflict

Answer: NO valid solution exists in this subtree (all rows in col 3 have
conflicts)
```

Alternative Example with Valid Solution: board = [3, 1, ?, ?]

If we instead had board = [3, 1, ?, ?]:

```
Col 3 options:
- Row 4: Safe? Check against (3,1) and (1,2)
  - Not same row ✓
  - Diagonal from (3,1)? |3-4| == |1-3|? 1 ≠ 2 ✓
  - Diagonal from (1,2)? |1-4| == |2-3|? 3 ≠ 1 ✗ NO - diagonal conflict

- Eventually trying all positions leads to board = [3,1,4,2] ✓ VALID SOLUTION
```

Running Time

- **Worst Case**: $O(n!)$ - must explore all permutations
- **Recurrence**: $T(n) = n \cdot T(n-1) + O(n)$
   - At each level, try up to n positions
   - Each position requires $O(n)$ safety check

  - Recurse to next column (n-1 columns remaining)
- **With Pruning**: Much better in practice, but still exponential

Key Insights

- **Backtracking** explores search space systematically
- **Pruning** eliminates invalid branches early
- **Constraint checking** prevents exploring doomed subtrees
- Position early in tree affects search tree size dramatically

Algorithm Design Techniques Summary

Technique	Strategy	Key Characteristics	Examples
Divide-and-Conquer	Break into subproblems, solve recursively, combine	Independent subproblems; T(n) = aT(n/b) + f(n)	Merge Sort, Binary Search, Karatsuba
Greedy	Make locally optimal choice at each step	Never backtracks; must prove correctness; efficient	Dijkstra's, Interval Scheduling, Connect Sticks
Backtracking	Build incrementally, backtrack when invalid	Explores search tree; abandons bad paths early	N-Queens, Sudoku, Graph Coloring

Heap Data Structure

Heap Properties

Min-Heap Property:

- Parent is smaller than or equal to children
- Smallest element at root (index 0 or 1)

Max-Heap Property:

- Parent is greater than or equal to children
- Largest element at root

Heap Violations

Checking Min-Heap Property:

- For each node with index i (up to heapSize-1):
  - Check if key[i] ≤ key[left_child(i)] (if left child exists)
  - Check if key[i] ≤ key[right_child(i)] (if right child exists)
- **Violation**: Parent is larger than one or more children

Important: Only check nodes within heapSize

- Elements beyond heapSize are not part of the heap

Heap Operations

Insert: $O(\log n)$

- Add element at end
- Bubble up to restore heap property

Extract-Min/Max: $O(\log n)$

- Remove root
- Move last element to root
- Bubble down to restore heap property

Decrease-Key: $O(\log n)$

- Reduce key value of element
- Bubble up to restore heap property

Build-Heap: $O(n)$

- Convert unordered array to heap
- Heapify from bottom up

Quick Reference Formulas

Graph Algorithms

- **BFS/DFS Time**: $O(V + E)$
- **Dijkstra Time**: $O((V + E) \log V)$ with binary heap
- **BFS Space**: $O(V)$ for queue
- **DFS Space**: $O(V)$ for stack/recursion

Karatsuba

- **Standard Multiplication**: $\Theta(n^2)$
- **Karatsuba**: $\Theta(n^{\log_2(3)}) \approx \Theta(n^{1.585})$
- **Recurrence**: $T(n) = 3T(n/2) + \Theta(n)$

N-Queens

- **Running Time**: $O(n!)$
- **Recurrence**: $T(n) = n \cdot T(n-1) + O(n)$

Heap Indexing (0-based)

- **Parent**: $\lfloor (i-1)/2 \rfloor$
- **Left Child**: $2i + 1$
- **Right Child**: $2i + 2$

Heap Indexing (1-based)

- **Parent**: $\lfloor i/2 \rfloor$
- **Left Child**: $2i$
- **Right Child**: $2i + 1$

Course content developed by Declan Gray-Mullen for WNEU with Claude