CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-1

Khwaab Thareja — N15911999 — kt3180

October 2024

Problem 2-1. Exercises on graph searches

The following problem parts are unrelated.

(a)What changes about the details of BFS if we represent the graph by an adjacency matrix instead of an adjacency list? What is the running time of the modified adjacency matrix-based algorithm? Explain your answer—don't just state a bound.

(b) Give a counterexample to the following conjecture: if there is a path from u to v in a directed graph G, then any depth-first search must result in $v.s \leq u.f$, where $v.s$ denotes the start time of v and $u.f$ denotes the finishing time of u.

There are counterexamples as small as 3 vertices. The smaller and more concise the counterexample, the better. If your example graph is too large, you may lose some credit. Note that you must explain why your counterexample is indeed a counterexample to the conjecture, i.e., you should explain how DFS would execute.

(c) You're presented as input an $n \times n$ 2D grid $G[1..n][1..n]$ of cells which are either clear (white) or blocked (black), modeling a maze where you can move horizontally or vertically (not diagonally) through white squares.

Describe and analyze a good algorithm for determining if there's a path from a given starting cell $s = (i, j)$ to a given ending cell $t = (i', j')$. You may apply an algorithm that we've learned as a black box. Note however that the problem is not stated as a graph — if you want to apply a graph algorithm, you would have to describe what the corresponding graph is.

For example, for this grid with displayed locations s and t, the answer should be "No." Your runtime bound should be a function of n, where the grid consists of n^2 cells.

(d) There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n wrestlers and a list of r pairs of wrestlers for which there are rivalries.

Give an $O(n + r)$-time algorithm to determine whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

Note that this problem does not explicitly provide a graph as input — your solution would need to specify what the graph would be.

Answer a:

To understand the changes, we would compare both the adjacency matrix and adjacency list based upon their features of how they are represented and how we check out the neighbors of a node in both forms.

- In an adjacency list, finding the neighbors of a vertex is straightforward since you can directly access the list of adjacent vertices. Iterating through them takes $O(\deg(v))$, where deg(v) is the number of neighbors (degree) of vertex v.
- Checking for neighbors is less efficient because you have to examine the entire row corresponding to the current vertex. This takes $O(V)$ time for each vertex.

We can also compare the adjacency list and matrix based upon their representation:

- In an **Adjacency List**, each vertex is linked to a list of its neighboring vertices. The space required for this representation is proportional to $O(V + E)$, where V is the number of vertices and E is the number of edges.
- An **Adjacency Matrix**, in contrast, is a $V \times V$ grid where matrix$[i][j] = 1$ indicates an edge between vertex i and vertex j, and matrix$[i][j] = 0$ means there's no edge. This method uses $O(V^2)$ space regardless of the actual number of edges.

Both the adjacency list and the adjacency matrix also differ in time complexity.

To look up the neighbors of vertex u in an adjacency list, it takes $O(\deg(u))$, and in total for all vertices, it takes $O(V+E)$ since both the edge and the vertex will be visited once.

However, in the adjacency matrix version, for each vertex u, you must check every vertex v to see if there is an edge from u to v by inspecting matrix$[u][v]$. This check must be done for all V vertices, making the total time complexity for exploring neighbors $O(V^2)$.

Therefore, The BFS algorithm is slower when using an adjacency matrix, especially if the graph has only a few edges compared to the number of vertices. This is because, with an adjacency list, BFS only looks at the actual edges between connected vertices, which makes it faster. On the other hand, the adjacency matrix requires BFS to check every possible connection between all pairs of vertices, even when most of them aren't connected. This extra work makes it take more time, resulting in a slower performance overall.

Answer b:

We can have a counterexample with as small as 3 vertices.

As per the conjecture: if there is a path from vertex u to vertex v in a directed graph G, then any depth-first search (DFS) must result in $v.s \leq u.f$, where $v.s$ is the start time of vertex v and $u.f$ is the finishing time of vertex u.

Let us consider a case where: consider

$$u \to v$$

$$u \to w$$

$$v \to u$$

Also show in the figure.

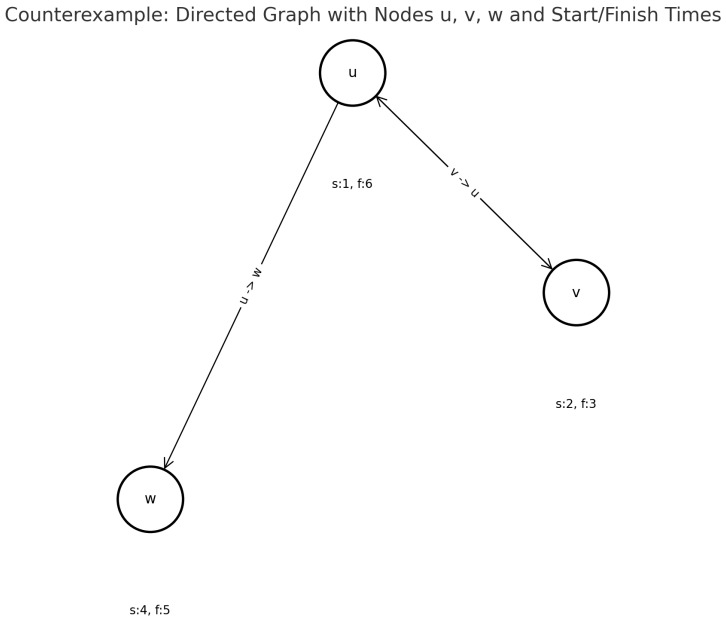

Figure 1: Counterexample: Directed Graph with Nodes u, v, w and Start/Finish Times

DFS begins at u, so the start time for u is $u.s = 1$. DFS explores the outgoing edges of u and first moves to vertex v.

DFS visits v, so the start time for v is $v.s = 2$. From v, DFS attempts to explore the edge

$$v \to u$$

but since u has already been visited, DFS backtracks and finishes exploring v. The finish time for v is $v.f = 3$.

After finishing vertex v, DFS backtracks to u and now moves to the next neighbor of u, which is w. The start time for w is $w.s = 4$. Since w has no further neighbors, DFS finishes exploring w, and its finish time is $w.f = 5$.

Finally, DFS finishes exploring u, and its finish time is $u.f = 6$.

Start and Finish Times Summary:

$$u.s = 1,\ u.f = 6$$

$$v.s = 2,\ v.f = 3$$

$$w.s = 4,\ w.f = 5$$

There is a path from $v \to w$, but contradicting the conjecture, we have $v.f = 3$ and $w.s = 4$. Clearly, $v.f < w.f$, so This demonstrates that the conjecture does not always hold in certain directed graphs.

Answer c:

We can move horizontally or vertically between white cells, and the goal is to return either "Yes" (if a path exists) or "No" (if no path exists).

To solve this, we can treat the grid as a graph, where:

- Each cell is a vertex.
- There is an edge between two vertices if their corresponding cells are adjacent (up, down, left, or right) and both are white.

To find whether a path exists, we can use either Breadth-First Search (BFS) or Depth-First Search (DFS). However, BFS is generally more optimized for this kind of problem. Here's why:

- Breadth-First Search (BFS) explores all neighboring cells level by level, ensuring that the first time it reaches the target t, it has found the shortest path.
- BFS is more systematic and efficient in grid traversal problems, where paths need to be explored in an orderly fashion. The time complexity of BFS is $O(n^2)$, where $n \times n$ is the size of the grid, as each cell is processed once.
- Depth-First Search (DFS), on the other hand, explores deeper paths first and only backtracks when it reaches dead ends.
- While DFS can also solve the problem, it may be less efficient because it can explore inefficient paths first. DFS doesn't guarantee the shortest path, and it may take longer in some cases compared to BFS.
- In this case, BFS is the more optimal choice because it efficiently finds the path (if one exists) and guarantees the shortest path. Therefore, we can conclude that BFS would be the preferred algorithm for determining if a path exists from s to t in the grid.
- Thus, the recommended approach is to apply BFS to explore all possible paths from s to t. If BFS reaches t, the answer is "Yes"; otherwise, it's "No." The time complexity of this approach is $O(n^2)$.

Therefore, Using BFS, we have the algorithm as stated below in which we take a 2D grid of N x N. We have starting cell s as (i,j) and will return yes if a path exists from s to t (i',j'). We will initialise an empty queue and an empty set to keep track of all the nodes that are visited and as we traverse through the matrix, we will check the all the neighbors of current node which are permissible to check as per the input matrix stating whether the neighboring node is white or black, unless we either reach the target node or the BFS queue gets empty

Algorithm 1 traversing the grid from s to t

Input: 2D grid G of size $n \times n$, starting cell $s = (i, j)$, target cell $t = (i', j')$
Output: Return "Yes" if a path exists from s to t, otherwise return "No"
directions $\leftarrow [(0, 1), (0, -1), (1, 0), (-1, 0)]$ {Possible movements: right, left, down, up}
queue $\leftarrow$ empty queue {Initialize the BFS queue}
visited $\leftarrow$ empty set {Initialize the visited set to keep track of explored cells}
enqueue $queue, s$ {Enqueue the start cell}
add s to visited {Mark the start cell as visited}
while queue is not empty **do**
  current $\leftarrow$ dequeue(queue) {Dequeue the next cell to explore}
  if current = t **then**
    return "Yes"
  end if
  for each direction $\in$ directions **do**
    neighbor_x $\leftarrow$ current_x + direction_x
    neighbor_y $\leftarrow$ current_y + direction_y
    if $0 \leq$ neighbor_x $< n$ and $0 \leq$ neighbor_y $< n$ and $G[neighbor\_x][neighbor\_y]$ = 'white' and $(neighbor\_x, neighbor\_y) \notin$ visited **then**
      enqueue $queue, (neighbor\_x, neighbor\_y)$
      add $(neighbor\_x, neighbor\_y)$ to visited
    end if
  end for
end while
return "No" {If the queue is empty and the target was not reached, return "No"} =0

We start by adding the starting cell s to the BFS queue. This queue will keep track of cells to explore, and we also use a set to mark cells as visited so we don't check the same cell twice.

Each time, we take a cell out of the queue and see if it's the target cell t. If it is, we return "Yes" because we've found a path. If it's not, we check its neighbors (cells next to it that are white and inside the grid). Any valid neighbors are added to the queue, and we mark them as visited.

The BFS continues until the queue is empty. If we haven't reached the target cell by then, we return "No," meaning there's no path to the target.

Answer d:

To solve this problem, we need to think of the wrestlers and their rivalries as a graph. While the problem doesn't directly give us a graph, we can easily create one from the input, where each wrestler is a **vertex**, and each rivalry between two wrestlers is an **edge**.

We will consider each wrestler as a vertex (or node) in the graph.

If two wrestlers are rivals, we connect them with an edge between their respective vertices. Since rivalries are mutual, this is an **undirected graph**.

We need to figure out if we can split the wrestlers into two groups — good guys and bad guys — such that each rivalry exists between a good guy and a bad guy. This is essentially asking if the graph is **bipartite**, meaning it can be divided into two distinct sets of vertices without any edges connecting vertices in the same set.

Therefore, We can use Breadth-First Search (BFS) to solve this problem efficiently.

1. We'll create an **adjacency list** to represent the graph, where each vertex (wrestler) points to the vertices it is connected to (its rivals).

2. Start at an unvisited wrestler, assigning them to one group (let's call them "good guys"). Then, we assign all their direct rivals to the other group (the "bad guys"). From there, continue this process, alternating between groups, using BFS to explore the graph. If we ever find a situation where two rivals are placed in the same group, we know that it's impossible to split the wrestlers the way we want.

3. It's possible that not all wrestlers are directly connected (the graph might be made up of multiple components). To handle this, we perform BFS on all unvisited wrestlers, ensuring that all wrestlers are checked.

Pseudocode:

```
function WrestlerGroups(n, rivalries):

    # Create empty adjacency list
    graph = [[] for i in range(n)]

    # graph from the rivalries list
    for rivalry in rivalries:
        u, v = rivalry
        graph[u].append(v)
        graph[v].append(u)

    # Initialize a list to store the group assignments
```

```
group = [-1] * n  # -1 = wrestler not in group yet

# BFS function to assign groups
function bfs(start):
    queue = [start]
    group[start] = 0  # initially first wrestler to group 0 (good guys)

    while queue is not empty:
        current = queue.pop(0)  # Dequeue the next wrestler

        # Explore all the rivals (neighbors)
        for neighbor in graph[current]:
            if group[neighbor] == -1:  # If the neighbor not in group
                group[neighbor] = 1 - group[current]  # put in opposite group
                queue.append(neighbor)
            elif group[neighbor] == group[current]:
                return False  # If rivals in same group, return False
    return True

# Check all wrestlers
for wrestler in range(n):
    if group[wrestler] == -1:  # If wrestler not visited
        if not bfs(wrestler):  # Perform BFS,check for conflicts
            return "No"  # If conflict, return No

return "Yes"  # If no conflicts, return Yes
```

We first build the graph using an adjacency list. This list stores the rivals of each wrestler. For example, if wrestler 1 is a rival of wrestler 2, we add 2 to the list of wrestler 1, and 1 to the list of wrestler 2.

Using BFS, we start with an unvisited wrestler and assign them to a group (either 0 for "good guys" or 1 for "bad guys"). We then assign their direct rivals to the opposite group.

As we continue the BFS, if we ever find two rivals assigned to the same group, we immediately know that the split isn't possible, and we return "No". If we finish the BFS and successfully assign all wrestlers without conflicts, we return "Yes".

If some wrestlers are disconnected from others, we perform BFS from any unvisited wrestler to ensure that every wrestler is checked.

Time Complexity:

Constructing the adjacency list takes $O(r)$, where r is the number of rivalries. BFS runs in $O(n+r)$, where n is the number of wrestlers and r is the number of rivalries. So, the total time complexity is $O(n+r)$, which is efficient and meets the problem's requirements.

Problem 2-2. Exercises on Strongly Connected Components

The following problem parts are unrelated.

(a) Describe an algorithm to compute the transpose of a directed graph, given in adjacency list representation, in $O(V+E)$ time. In this case, I want you to also provide pseudocode.

(b) Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second DFS and scanned the vertices in increasing finishing times. Does this simpler algorithm always produce correct results? If yes, justify your answer. If no, provide and explain a counterexample.

(c) Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (i) G' has the same strongly connected components as G, (ii) G' has the same component graph as G, and (iii) E' is as small as possible. Describe a fast algorithm to compute G' and analyze its running time.

Answer 2.a:

To find the transpose of a directed graph, we need to reverse all the edges in the graph. This means if there is an edge from vertex u to vertex v in the original graph, we create an edge from v to u in the transpose graph.

- First, we create a new adjacency list for the transpose graph. This list will have the same number of vertices as the original graph, but no edges yet.
- For each vertex u in the original graph, we check all the vertices v it connects to. For each edge $u \to v$, we add the edge $v \to u$ to the transpose graph.
- After reversing all the edges, we return the new adjacency list, which now represents the transpose graph.

Time Complexity:

The algorithm runs in $O(V + E)$, where:

- V is the number of vertices.
- E is the number of edges.

We process every vertex once and every edge once, which makes the time complexity efficient.

Pseudocode:

```
function computeTransposeGraph(V, adjList):
    # Initialize an empty list where we can have the transpose
    transposeAdjList = [[] for _ in range(V)]

    # Loop through each vertex and reverse the edges
    for u in range(V):
        for v in adjList[u]:
            # For each edge u -> v, add v -> u in the transpose graph
            transposeAdjList[v].append(u)

    return transposeAdjList
```

Example:

Original graph edges:

$$a \to b$$

$$a \to c$$

$$b \to c$$

$$c \to d$$

Transpose graph:

$$b \to a$$

$$c \to a, b$$

$$d \to c$$

Answer 2.b:

Professor Bacon suggests a simplified version of Kosaraju's algorithm for finding SCC . His proposal involves using the original graph for the second DFS (instead of the transpose graph) and processing vertices in increasing order of their finishing times. The goal is to determine whether this modification produces the correct results.

Kosaraju's algorithm for finding SCCs follows these steps:

1. Perform a DFS on the original graph G to compute the **finishing times** of all vertices.
2. Reverse all the edges of G to form the transpose graph G^T.
3. Perform another DFS on G^T, but this time process the vertices in **decreasing order** of their finishing times from the first DFS.
4. Each DFS tree from the second DFS represents an SCC.

The purpose of using the **transpose graph** is to ensure that we capture the backward reachability of vertices, which is essential for correctly identifying SCCs.

In Professor Bacon's version, he suggests:

1. Running the second DFS on the **original graph** G, not the transpose graph.
2. Processing the vertices in **increasing order** of their finishing times from the first DFS.

However, The simpler approach fails because it breaks the fundamental requirement of reaching the node from both forward and backward direction in strongly connected components.

- it is essential to reach the nodes backwards as well to identify SCC's because all vertices in an SCC must be reachable from each other in both directions. The **transpose graph** G^T helps to reverse the edges and ensure that we explore all vertices reachable in the reverse direction, which is why the second DFS is performed on the transpose graph.
- **Order of processing vertices**: In the correct algorithm, we process vertices in **decreasing order** of finishing times, ensuring that we fully explore an SCC before moving on to the next one. Processing vertices in **increasing order** disrupts this, leading to incorrect groupings.

Counterexample:

Consider the following directed graph:

$$v_1 \to v_2, \quad v_2 \to v_1, \quad v_1 \to v_3$$

1. **First DFS**: Assume the finishing times are:

$$f(v_3) = 1, \quad f(v_1) = 2, \quad f(v_2) = 3$$

2. **Second DFS on the original graph**: Process vertices in **increasing order** of finishing times: v_3, v_1, v_2.
3. Start with v_3, which forms its own SCC.
4. Then process v_1, visit v_2, and return to v_1, treating $\{v_1, v_2\}$ as a single SCC.
5. The algorithm incorrectly merges v_1, v_2, v_3 into one SCC.

Correct SCCs:

- $\{v_1, v_2\}$ is one SCC.
- $\{v_3\}$ is another SCC.

Therefore, Professor Bacon's simplified algorithm doesn't work correctly because:

1. It uses the original graph instead of the reversed (transpose) graph, which misses the ability to trace back connections between nodes.
2. It processes nodes in increasing order of finishing times, which messes up the logic needed to correctly identify strongly connected components (SCCs).

The correct approach, Kosaraju's algorithm, runs the second depth-first search (DFS) on the reversed graph and processes nodes in **decreasing order** of finishing times. This ensures that all SCCs are correctly found.

Answer 2.c:

We are given a directed graph $G = (V, E)$, and we need to create another graph $G' = (V, E')$ with the following conditions:

1. G' should have the same strongly connected components (SCCs) as G.

2. G' should maintain the same relationships between the SCCs as in G.
3. G' should have as few edges as possible.

To solve this, we have to follow two main steps:

1. Identify the SCCs of the graph G.
2. Create a new graph G' that captures the relationships between SCCs, ensuring that the number of edges is minimized.

We will use the following algorithm to find SCCs

1. Perform a DFS on the original graph G, and track the finishing times of the vertices.
2. Reverse all the edges in G to create the **transpose graph**.
3. Run a second **DFS** on the transpose graph, but process the vertices in the order of **decreasing finishing times** from the first DFS. Every time DFS completes, it identifies one SCC.

After identifying the SCCs, we create the component graph G' where:

- Each SCC from the original graph is represented as a single node in G'.
- For each edge in G, if the edge connects two vertices from different SCCs, we add a directed edge between their corresponding SCCs in G'.

We ensure that only one edge is added between each pair of SCCs to minimize the number of edges in G'.

Algorithm:

1. Use Kosaraju's algorithm to group vertices into SCCs.
2. Create an empty graph G', with one node for each SCC.
3. For every edge $u \to v$ in the original graph G, check if u and v belong to different SCCs.
4. If they do, add a directed edge between their corresponding SCCs in G', making sure to add only one edge between any two SCCs.

Pseudocode:

```
function computeComponentGraph(V, E):
    # Step 1: Find SCCs using Kosaraju's Algorithm
    SCCs = KosarajuSCC(V, E)  # Get the list of SCCs

    # Step 2: Create the component graph G'
    G' = createEmptyGraph(len(SCCs))  # Create an empty graph for the SCCs

    # Add edges between SCCs
    for each edge (u, v) in E:
        SCC_u = findSCC(u, SCCs)
        SCC_v = findSCC(v, SCCs)

        if SCC_u != SCC_v and not G'.hasEdge(SCC_u, SCC_v):
            G'.addEdge(SCC_u, SCC_v)  # Add the edge between the SCCs

    return G'  # Return the component graph
```

Time Complexity:

- Finding SCC takes $O(V + E)$ time because it involves two depth-first searches (DFS) and reversing the graph.
- For each edge in the original graph, we check whether it connects different SCCs. This also takes $O(V + E)$.

So, the total time complexity of the entire algorithm is $O(V + E)$, which is efficient for this problem.

Problem 2-3: Constructing a Minimum Stack and Queue

You are tasked with constructing a **Minimum Stack** and **Minimum Queue** that efficiently supports standard operations and a `min()` operation that returns the minimum element in constant time. For each part, you are required to:

- Provide the algorithm for the given task.
- Analyze the time complexity of each operation.

(a) Minimum Stack

Design a data structure called **Minimum Stack** that supports the following operations:

- `Push(x)`: Add element x to the top of the stack.
- `Pop()`: Remove the element from the top of the stack.
- `Top()`: Return the top element of the stack.
- `Min()`: Return the minimum element in the stack in constant time.

(b) Minimum Queue

Construct a **Minimum Queue** that supports:

- `Enqueue(x)`: Add element x to the end of the queue.
- `Dequeue()`: Remove the element from the front of the queue.
- `Min()`: Return the minimum element in the queue in constant time.

Answer 3a

To ensure that the *Min()* function works efficiently as a minimum Stack, we use two stacks:

1. A **main stack** to store all the elements.
2. A **min stack** to store the minimum values at each level of the main stack.

We need to create a *Minimum Stack* that supports the following operations:

- **Push(x)**:
  - Add x to the *main stack*.
  - If x is smaller than or equal to the current minimum, push it onto the *min stack* as well.
- **Pop()**:
  - Remove the top element from the *main stack*.
  - If that element matches the top of the *min stack*, remove it from the *min stack* as well.
- **Top()**:

  - Return the element on top of the *main stack*.

- **Min()**:
  - Return the top element of the *min stack*, which holds the current minimum.

Whenever a new element is added to the *main stack*, we check if it is smaller than or equal to the current minimum. If it is, it is also pushed onto the *min stack*. When an element is removed from the *main stack*, if it matches the minimum, we also remove it from the *min stack*.

Time Complexity:

- **Push(x)**: $O(1)$ – Adding elements to both stacks takes constant time.
- **Pop()**: $O(1)$ – Removing elements from both stacks also takes constant time.
- **Top()**: $O(1)$ – Retrieving the top element is constant time.
- **Min()**: $O(1)$ – The minimum element is always at the top of the *min stack*, so we can retrieve it in constant time.

Algorithm 2 Minimum Stack

```
0: Initialize: stack ← empty, minimum_stack ← empty
0: procedure PUSH(x)
0:     Push x onto stack
0:     if minimum_stack is empty or x ≤ minimum_stack.top() then
0:         Push x onto minimum_stack
0:     end if
0: end procedure
0: procedure POP
0:     if stack is not empty then
0:         if stack.top() == minimum_stack.top() then
0:             Pop from minimum_stack
0:         end if
0:         Pop from stack
0:     else
0:         return "Stack is empty"
0:     end if
0: end procedure
0: procedure TOP
0:     if stack is not empty then
0:         return stack.top()
0:     else
0:         return "Stack is empty"
0:     end if
0: end procedure
0: procedure MIN
0:     if minimum_stack is not empty then
0:         return minimum_stack.top()
0:     else
0:         return "Stack is empty"
0:     end if
0: end procedure
  =0
```

Answer 3.b

We need to create a *Minimum Queue*

For this, we'll use two queues:

1. A **main queue** to store all the elements.
2. A **min queue** to keep track of the minimum elements.

As we add elements to the main queue, we make sure that the min queue only keeps track of the smallest values. If a new element is smaller than the elements at the back of the min queue, we remove those larger elements. This way, the smallest value is always at the front of the min queue.

When removing elements from the main queue, we check if the front element matches the front of the min queue. If it does, we remove it from the min queue as well.

We will have the following operations:

- **Enqueue(x)**:
  - Add x to the back of the *main queue*.
  - If x is smaller than the elements at the back of the *min queue*, remove those larger elements.
  - Add x to the back of the *min queue*.
- **Dequeue()**:
  - Remove the front element from the *main queue*.
  - If it matches the front element of the *min queue*, remove it from the *min queue* too.
- **Min()**:
  - Return the front element of the *min queue*, which holds the current minimum.

Time Complexity:

- **Enqueue(x)**: $O(1)$ on average – Even though removing elements from the *min queue* takes some time, each element is removed only once, making it efficient.
- **Dequeue()**: $O(1)$ – Removing from both queues takes constant time.
- **Min()**: $O(1)$ – The minimum is always at the front of the *min queue*, so retrieving it is constant time.

Both the *Minimum Stack* and *Minimum Queue* use an additional stack or queue to keep track of the minimum elements, allowing all standard operations and retrieval of the minimum element to be done in constant time.

Algorithm 3 Minimum Queue

```
0: Initialize: queue ← empty, minimum_queue ← empty
0: procedure ENQUEUE(x)
0:     Enqueue x into queue
0:     while minimum_queue is not empty and minimum_queue.back() > x do
0:         Pop from the back of minimum_queue
0:     end while
0:     Enqueue x into minimum_queue
0: end procedure
0: procedure DEQUEUE
0:     if queue is not empty then
0:         if queue.front() == minimum_queue.front() then
0:             Dequeue from minimum_queue
0:         end if
0:         Dequeue from queue
0:     else
0:         return "Queue is empty"
0:     end if
0: end procedure
0: procedure MIN
0:     if minimum_queue is not empty then
0:         return minimum_queue.front()
0:     else
0:         return "Queue is empty"
0:     end if
0: end procedure=0
```

Problem 7: Analyzing the Performance of a Hash Table using Linear Probing for Collision Resolution

Assume that the hash function h is uniformly random, meaning that:

$$Pr[h(e) = i] = \frac{1}{m}$$

for any key e and any index $i \in \{0, 1, 2, \ldots, m-1\}$. We assume that n elements have already been inserted into the hash table, and let S represent the set of elements in the hash table. Hence, $|S| = n$. Define the load factor α as:

$$\alpha = \frac{n}{m}$$

where n is the number of inserted elements and m is the size of the hash table. Assume that α is a constant. We are interested in analyzing the relationship between the load factor α and the access time for a randomly chosen element from S.

The following algorithm describes how we insert an element into the hash table using linear probing with uniform hashing:

Algorithm 1: Insert with Linear Probing and Uniform Hashing

- Input: Key e, Hash table T of size m
- Output: Insert key e into T
- Step 1: $i \leftarrow h(e)$ (Hash the key e to index i)
- Step 2: While $T[i] \neq$ None, do:
  - $i \leftarrow (i + 1) \mod m$
- Step 3: $T[i] \leftarrow e$ (Insert the key e into the empty slot)
- Step 4: Return

To better understand the average cost of accessing elements, consider the following scenario. For each slot i in the hash table, we initialize a counter $C_i = 0$. Whenever we insert an element, we query the hash function to obtain a random slot, and if the slot is occupied, we move to the next slot (using linear probing). We increment the counters for all slots we touch during the probing process. For instance, if the hash function gives us index 4, but slots 4, 5, and 6 are occupied while slot 7 is empty, we insert the element in slot 7 and increment C_4, C_5, C_6, C_7 by 1.

(a)

After finishing all insertions, the counters C_i follow a certain distribution. What is the expected access time for retrieving a random element from the hash table in terms of the counters C_i? Specifically, derive the expected access time based on the number of slots probed during an insertion.

(b)

Let $B_j(i)$ be the indicator that after n insertions, $C_i = j$, meaning that slot i has been touched j times. More formally, $B_j(i) = 1$ if the slot i is touched j times after all insertions, and $B_j(i) = 0$ otherwise. How are the following probabilities related?

$$Pr[B_j(i) = 1] \quad \text{and} \quad Pr[B_j(i+1) = 1]$$

Use this relation to establish a connection between $E[C_i]$ and $E[C_{i-1}]$.

(c)

Let $N(i) = |\{e \in S \mid h(e) = i\}|$ represent the number of elements whose hash function value equals i. In other words, $N(i)$ is the number of elements that are hashed to the i-th slot of the hash table. For large enough n and m, show that:

$$Pr[N(i) = k] \approx \frac{e^{-\alpha}\alpha^k}{k!}$$

where $\alpha = \frac{n}{m}$ is the load factor, and e is the natural base of the exponential function.

(d)

Consider the event $B_j(i) = 1$, which indicates that slot i is touched j times during the whole insertion process. For a fixed i, if $B_j(i)$ occurs, what can you infer about the states of slots $i-1$ and the number of elements hashed to slot $i-1$, denoted by $N(i-1)$? Let $q_j = Pr[B_j(i) = 1]$ be the probability that slot i is touched j times, and $p_k = Pr[N(i) = k]$ be the probability that k elements are hashed to slot i. Explain the equation:

$$q_j = q_0 p_j + q_1 p_{j-1} + q_2 p_{j-2} + \cdots + q_{j+1} p_0$$

Hint: Analyze how the fullness of slot i depends on the state of slot $i-1$ and the number of elements hashed to it.

(e)

Now assume that n is infinite. From the previous problem, we can derive a set of equations for q_j using the probabilities $q_{j+1}, q_j, \ldots, q_0$ and the corresponding probabilities $p_j, p_{j-1}, \ldots, p_0$, for $j \in [0, n]$. We are interested in computing the

expected value of C_i, which is the number of slots touched during the insertion of an element:

$$E[C_i] = \sum_{j=1}^{\infty} jq_j$$

To compute $\sum_{j=1}^{\infty} jq_j$, we will apply a less formally defined method using generating functions. To begin, let's focus on computing q_0. One approach to find q_0 is by using generating functions. Define the following generating function for the probabilities p_k:

$$f(x) = \sum_{k=0}^{\infty} p_k x^k$$

where $p_k = \frac{e^{-\alpha}\alpha^k}{k!}$. By expanding the function, show that $f(x)$ is a well-known result of Taylor expansion.

(f)

Now, define a new generating function for the sequence q_k:

$$g(x) = \sum_{k=0}^{\infty} q_k x^k$$

Given the previous problems and the structure of the system, show that $g(x)$ is given by:

$$g(x) = q_0 \frac{e^{\alpha x(x-1)}}{xe^{\alpha} - e^{\alpha x}}$$

(g)

Next, use L'Hôpital's rule to compute q_0 from the expression for $g(x)$. Specifically, evaluate the limit of $g(x)$ as $x \to 1$ to obtain q_0. What's q_0? Does it make sense?

(h)

Next, compute $g'(1)$. First, compute $g'(x)$ and then use L'Hôpital's rule to evaluate $g'(x)$ as $x \to 1$.

Answer 7.a

C_i keeps track of how many times we try to place items into a specific spot in the hash table before finding an empty slot. If multiple items are hashed to the same position, each attempt to place those items increases the counter for that particular spot.

C_i represents how many times each slot is probed during the insertion process.

The average expected access time, $E(\text{Access time})$, is calculated as:

$$E(\text{Access time}) = \frac{1}{n} \sum_{i=0}^{m-1} C_i \approx \frac{1}{1-\alpha}$$

where n is the number of elements inserted and m is the size of the hash table.

As the load factor $\alpha = \frac{n}{m}$ increases (meaning the table becomes fuller), more collisions occur, and items take longer to find an empty spot. Consequently, the expected access time increases as well. The new expected access time is given by:

$$E(\text{Access time}) \approx 1 + \frac{1}{n} \sum_{i=0}^{m-1} C_i$$

As the number of collisions increases, the access time becomes longer, since it takes more attempts to place an item into the hash table. This provides a clear explanation of how probing works in a hash table with linear probing, the effect of load factor, and how it influences access time.

Answer 7.b

In *linear probing*, if slot i is probed j times, slot $i-1$ must have been probed at least j times. Thus, the probability decreases as you move forward in the sequence:

$$P(B_j(i) = 1) \geq P(B_j(i+1) = 1)$$

This implies that earlier slots are more likely to be probed. Therefore, the expected number of probes at slot i is less than or equal to that at slot $i-1$:

$$E[C_i] \leq E[C_{i-1}]$$

This shows that earlier slots tend to experience more probes.

Hence solvng the equations as per requirement:

1. $$P(B_j(i) = 1) \geq P(B_j(i+1) = 1)$$

   Then

   $$E[C_i] \quad \text{and} \quad E[C_{i-1}]$$

2.
$$P(B_j(i) = 1) \geq P(B_j(i-1) = 1)$$
In linear probing, if slot i is probed j times, it means slot $i-1$ has been probed at least j times. That is, if there is a collision, then it moves to the next slot.

$$P(B_j(i) = 1) \leq P(B_j(i-1) = 1)$$
or
$$P(B_j(i+1) = 1) \leq P(B_j(i) = 1)$$
Slot $i+1$ is probed j times, so slot i must have been probed at least j times.

3.
$$E[C_i] = \sum_{j=1}^{n} (j \times P(B_j(i) = 1))$$
for j from 1 to n.

$$E[C_{i-1}] = \sum_{j=1}^{n} (j \times P(B_j(i-1) = 1))$$
Since
$$P(B_j(i) = 1) \leq P(B_j(i-1) = 1) \quad \text{for all } j,$$

$$E[C_i] \leq E[C_{i-1}]$$

Answer 7.c

This equation helps us figure out the chance that exactly k elements end up in a specific slot i in a hash table, shown as $N(i) = k$.

- The number of elements that go into slot i, called $N(i)$, follows a *binomial distribution*. This simply means:
  - n is the total number of elements we're adding to the hash table.
  - m is the number of slots in the table.
  - $p = \frac{1}{m}$ is the chance that any single element gets placed into slot i.

  The binomial formula helps us figure out how likely it is that exactly k elements land in slot i.
- The binomial distribution formula looks like this:
$$P(N(i) = k) = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k}$$

- When n (the number of elements) is large, we can simplify things using the *Poisson distribution*. In this case, $\lambda = np = \frac{n}{m} = \alpha$, where α is the average number of elements you expect in each slot.
- With the Poisson approximation, the probability that exactly k elements land in slot i is:
$$P(N(i) = k) = \frac{\alpha^k e^{-\alpha}}{k!}$$

This approximation makes it easier to do the math, especially when there are a lot of elements being inserted into the hash table.

Answer 7.d

The equation $q_j = q_0 p_j + q_1 p_j + q_2 p_{j-1} + \cdots + q_{j+1} p_0$ explains how slot i in a hash table is probed j times during an insertion. This depends on the state of slot $i-1$ and how many elements hash to slot i.

- Slot $i-1$ is touched 0 times (q_0), and j elements hash directly to slot i (p_j).
- Slot $i-1$ is touched 1 time (q_1), and j elements hash to slot i (p_j).
- Slot $i-1$ is touched 2 times (q_2), and $j-1$ elements hash to slot i (p_{j-1}).
- Slot $i-1$ is touched $j+1$ times (q_{j+1}), and 0 elements hash to slot i (p_0).

This equation shows the sum of all probabilities for the ways slot i could be probed j times:

- It accounts for how slot $i-1$ is touched and how many elements hash to slot i.
- It considers the "spillover" effect from slot $i-1$ to i during linear probing.
- This recursive relationship illustrates how collisions and probing propagate through the table.
- It helps understand how often each slot is probed, forming the foundation for analyzing hash table performance.

Answer 7.e

We define a function $f(x)$ to calculate the probability that a certain number of elements land in a specific slot. This function is written as:

$$f(x) = \sum_{k=0}^{\infty} p_k x^k$$

where p_k is the probability that exactly k elements are placed in a slot, given by:

$$p_k = \frac{e^{-\alpha}\alpha^k}{k!}$$

Substituting this into the generating function:

$$f(x) = \sum_{k=0}^{\infty} \frac{e^{-\alpha}\alpha^k}{k!} x^k$$

Factoring out the constant $e^{-\alpha}$:

$$f(x) = e^{-\alpha} \sum_{k=0}^{\infty} \frac{(\alpha x)^k}{k!}$$

This sum is recognized as the Taylor series for $e^{\alpha x}$, so we can simplify:

$$f(x) = e^{-\alpha} e^{\alpha x} = e^{\alpha(x-1)}$$

This result gives us a compact form that makes it easier to analyze how elements are distributed in a hash table and how often different slots are probed.

Answer 7.f

We want to show that the generating function $g(x)$ is:

$$g(x) = \frac{q_0 e^{\alpha x}(x-1)}{xe^{\alpha} - e^{\alpha x}}$$

We know $f(x) = e^{\alpha(x-1)}$ is the generating function for p_k.
The equation $q_j = q_0 p_j + q_1 p_j + q_2 p_{j-1} + \dots$ can be expressed as:

$$g(x) = f(x)(q_0 + xq_1 + x^2 q_2 + \dots) = f(x)g(x)$$

This gives us the equation:

$$g(x) = e^{\alpha(x-1)} g(x)$$

After rearranging and simplifying:

$$g(x) = \frac{q_0 e^{\alpha x}(x-1)}{xe^{\alpha} - e^{\alpha x}}$$

This result shows the generating function $g(x)$, which captures the relationship between the probabilities q_k and p_k, and describes how probing works in the hash table.

Answer 7.g

We want to find q_0, which is the probability that a slot in the hash table is never touched during probing, using L'Hôpital's rule.

$g(x)$ is a function that helps us understand how often different slots in the hash table are probed. q_0 specifically tells us how likely it is that a slot is never touched.

We are looking for:

$$q_0 = \lim_{x \to 1} g(x)$$

If this limit gives us an indeterminate form (like $\frac{0}{0}$), we can apply L'Hôpital's rule. This rule lets us find the limit by taking the derivatives of both the top and bottom parts of $g(x)$.

After applying L'Hôpital's rule and differentiating both parts, we calculate the limit when x approaches 1. This gives us q_0, which should be a positive number.

- q_0 shows the chance that a slot is never probed.
- When the table isn't full (low load factor), q_0 is higher, meaning many slots aren't touched.
- As the table fills up (high load factor), q_0 gets smaller because more slots are being probed due to collisions.

In summary, q_0 helps us understand how likely it is that a slot in the hash table remains empty, and it decreases as the table becomes fuller.

Answer 7.h

We have,

$$g(x) = \sum_{j=0}^{\infty} q_j x^j$$

Here, q_j represents the probability that a slot is probed exactly j times.
We take the derivative of $g(x)$:

$$g'(x) = \frac{d}{dx}\left(\sum_{j=0}^{\infty} q_j x^j\right) = \sum_{j=1}^{\infty} j q_j x^{j-1}$$

This gives us the derivative of $g(x)$.
To compute $g'(1)$, we substitute $x = 1$:

$$g'(1) = \sum_{j=1}^{\infty} j q_j$$

This sum represents the expected number of probes, where each probe count j is weighted by how likely it is to occur.

- $g'(1)$ gives the expected number of probes needed to access a slot in the hash table.
- If there are more collisions (where multiple elements want the same slot), the value of $g'(1)$ increases, meaning more probes are needed to find an open slot.

In short, $g'(1)$ tells us how many probes we expect, and as the table gets fuller, more probes are required.