Dylan Phoutthavong

April 18th, 2025

CSCI 3412

HW5

Q 1) (5 points) Answer the following case scenario

Case Scenario:

You are in charge of an IT department maintaining network infrastructure in a large global corporation. The corporation's network consists of numerous interconnected servers spread across many different locations worldwide. To simplify, imagine the network of servers as a directed graph where nodes represent servers and edges represent direct connections between them.

Due to recent security vulnerabilities discovered in the networking software, it's crucial to apply patches sequentially for all the servers in the same subtree before patching others in the other subtrees. The challenge is that a server is patched in a bottom-up fashion, meaning that a server is patched only after all servers that directly connect to it (i.e., from which data flows to it) have been patched to prevent potential system crashes.

Questions: Provide your answer in English (1 point for each question)

a) Explain which search algorithm you will use and how you will utilize the algorithm to determine a safe sequence in which to apply patches to the servers. (***Hint***: Two things to consider: 1) Tree search algorithm, 2) tree traversal algorithm.)

Algorithm: Post-order Depth-First Search (DFS) Traversal

Reasoning:

- Since the network is modeled as a **direct tree/graph**, where patching must happen **bottom-up** (i.e., children before parents), **post-order traversal** is ideal.
- Post-order ensures all **child nodes** (**connected servers**) are visited and patched **before** their parent node is considered

How to use it:

- Perform a DFS starting from leaf servers.

- Visit and patch **all child nodes first**, then patch the **current node**.
- This guarantees that a server is not patched until all the servers it depends on (those that send it data) are patched.

b) Describe a pseudo-code for your approach.

```
function patchServer(server):
  if server is already patched:
    return
  for each child in server.children:
    patchServer(child)
  applyPatch(server)
  mark server as patched
```

c) As discussed in the class, describe any potential challenges or pitfalls using the search algorithm for this patch application and how you might address them.

Challenge 1: **Cycles** in the graph (if not a true tree)

- **Solution**: Use a visited set to prevent infinite loops

Challenge 2: **Unreachable servers** or **missing dependency info**

- **Solution**: Preprocess the graph to detect disconnected components or missing links.

Challenge 3: **Large networks** may cause stack overflow with recursive DFS.

- **Solution**: Implement an **iterative DFS** using a stack to avoid deep recursion.

d) While you are traversing for a path in the network leading to a group of servers. You should halt your traversal immediately if any one of the pre-specified target servers has already been patched. Provide your algorithm in such a situation and discuss any potential challenges.

Modified Approach:

- Add a check at the beginning of the traversal to see if **a target server has already been patched**.
- If so, **halt traversal** immediately and avoid patching dependent nodes to prevents inconsistency.

Pseudo code:

```
function patchServerWithCheck(server, targetServers):
  if any targetServer in targetServers is already patched:
    return "Halt traversal"
  if server is already patched:
    return
  for each child in server.children:
    patchServerWithCheck(child, targetServers)
  applyPatch(server)
  mark server as patched
```

Challenges:

- May halt traversal prematurely, requiring coordination to **resume later** from safe entry points.
- Need **synchronization** if multiple traversal instances are running in parallel.

e) Given that servers in this network also handle mission-critical user requests, abruptly patching a server can disrupt the user experience due to downtime. Introduce a mechanism in your traversal algorithm that ensures a server can be patched only if its load is below a certain threshold, thus minimizing user disruptions. If a server is above this threshold, the algorithm should temporarily skip it and continue with others, attempting to return to the skipped server later. Provide a modified pseudo-code for this approach and discuss potential challenges.

Modified Behavior:

- Before patching a server, check if its **current load < load threshold**.
- If the load is too high, **skip temporarily**, add it to a retry list, and attempt to patch later.

Pseudo code:

```
function patchWithLoadCheck(server, loadThreshold):
  if server is already patched:
    return
  for each child in server.children:
    patchWithLoadCheck(child, loadThreshold)
  if server.load < loadThreshold:
    applyPatch(server)
    mark server as patched
  else:
    add server to retryList
function retrySkippedServers(retryList, loadThreshold):
  for server in retryList:
    if server.load < loadThreshold:
      applyPatch(server)
      mark server as patched
    else:
      log "Server load still too high, try again later"
```

Challenges:

- **Retrying logic** must include timeout or retry limits.

- Server load may **fluctuate** unpredictably-introduce wait times or backoff strategies.

Q 2) (10 points): DP exercises

Q2-1) [Total 6 points] California vs. Carolina

Solve the following problems **on paper by hand** and submit the images of the paper in **one** PDF file.

1) LCS [3 points]: Build a 2-D DP table to find the LCS between the two words above (California and Carolina)

Recurrence Relation:

If California[i-1] == Carolina[j-1]:

dp[i][j] = dp[i-1][j-1] + 1

Else:

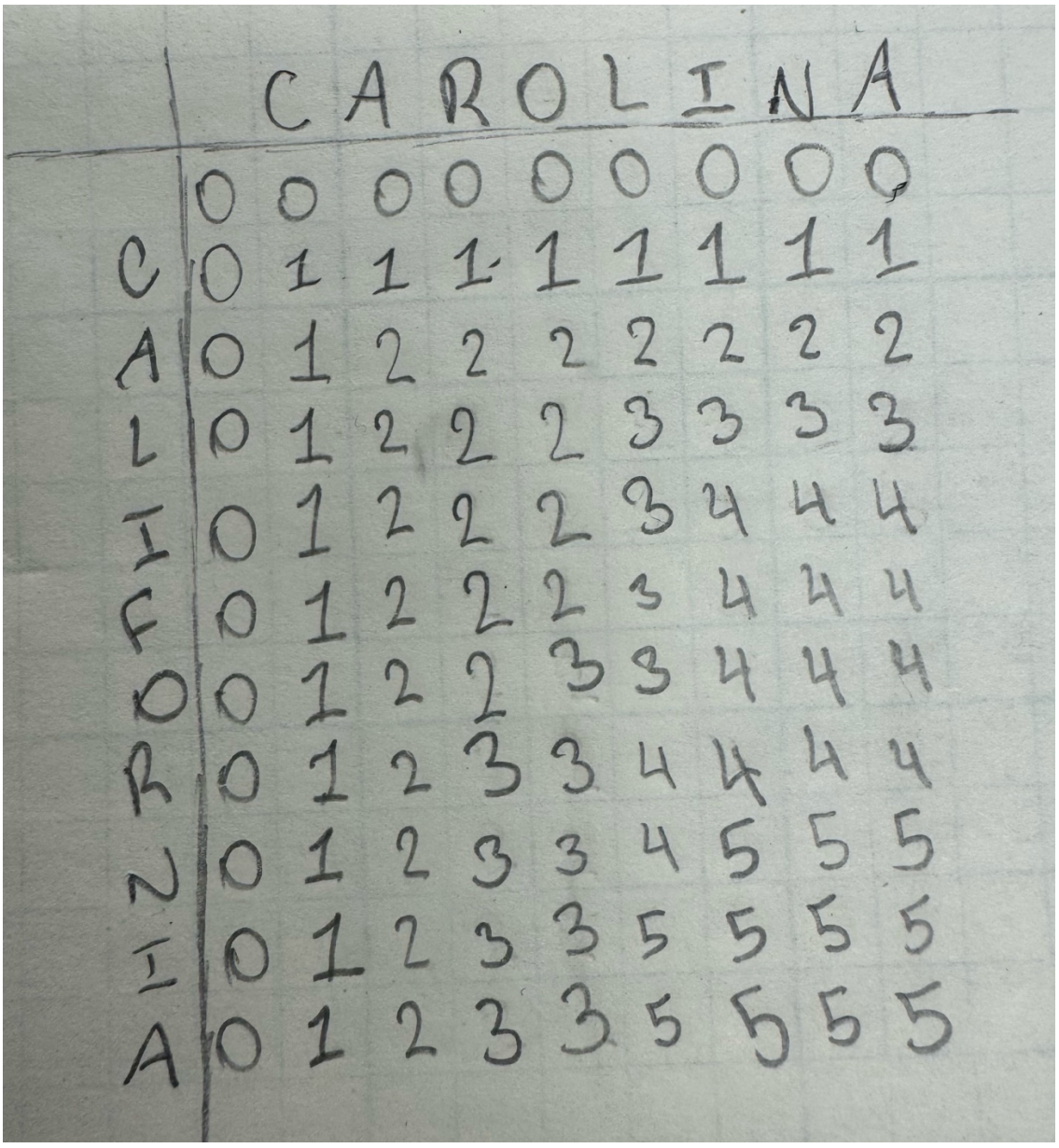
dp[i][j] = max(dp[i-1][j], dp[i][j-1])

		C	A	R	O	L	I	N	A
	0	0	0	0	0	0	0	0	0
C	0	1	1	1	1	1	1	1	1
A	0	1	2	2	2	2	2	2	2
L	0	1	2	2	2	3	3	3	3
I	0	1	2	2	2	3	4	4	4
F	0	1	2	2	2	3	4	4	4
O	0	1	2	2	3	3	4	4	4

R	0	1	2	3	3	3	4	4	4
N	0	1	2	3	3	3	4	5	5
I	0	1	2	3	3	3	5	5	5
A	0	1	2	3	3	3	5	5	5

Final LCS value: dp[10][8] = 6

One possible LCS: CALINA



2) Edit Distance [3 points]: Build a 2-D DP table to find the minimum Edit Distance between the two words above

Recurrence Relation:

If California[i-1] == Carolina[j-1]:

dp[i][j] = dp[i-1][j-1]

Else:

dp[i][j] = 1 + min(

dp[i-1][j], // delete

dp[i][j-1], // insert

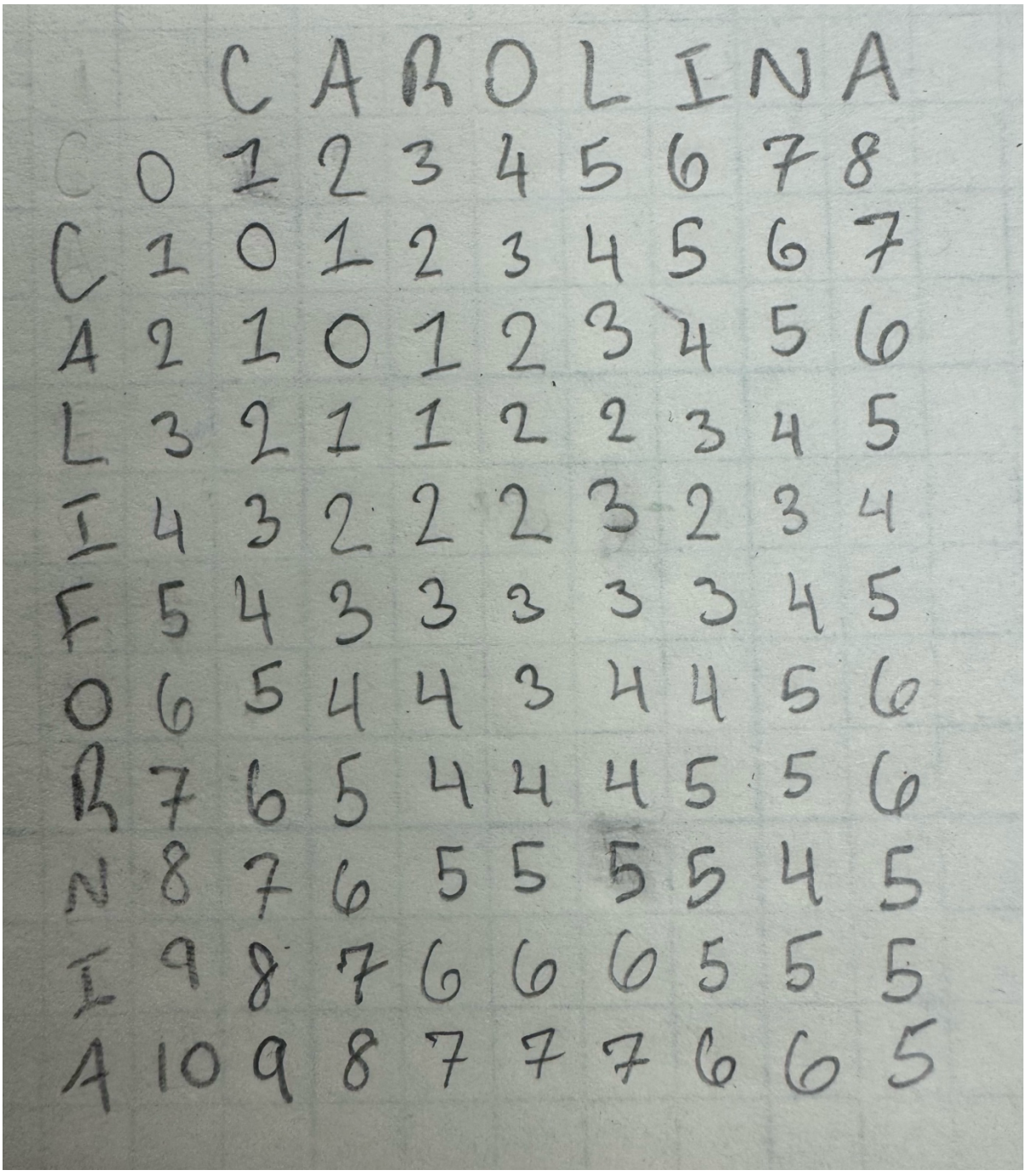
dp[i-1][j-1] // substitute

)

		C	A	R	O	L	I	N	A
	0	1	2	3	4	5	6	7	8
C	1	0	1	2	3	4	5	6	7
A	2	1	0	1	2	3	4	5	6
L	3	2	1	1	2	2	3	4	5
I	4	3	2	2	2	3	2	3	4
F	5	4	3	3	3	3	3	4	5
O	6	5	4	4	3	4	4	5	6
R	7	6	5	4	4	4	5	5	6

N	8	7	6	5	5	5	5	4	5
I	9	8	7	6	6	6	5	5	5
A	10	9	8	7	7	7	6	6	5

Minimum Edit Distance: dp[10][8] = 5



Q2-2) [Total 4 points] Bin Packing Problem Links to an external site.https://en.wikipedia.org/wiki/Bin_packing_problemLinks to an external site.

Question: Why is Dynamic Programming (DP) generally considered impractical for solving the Bin Packing Problem in real-world scenarios? Support your reasoning by discussing how the problem conflicts with the two main characteristics of dynamic programming approaches discussed in the class.

Dynamic Programming is generally considered **impractical** for solving Bin Packing Problem in real-world scenarios due to two key conflicts with the **characteristics of DP**:

1. **Optimal Substructure Violation**:
    a. In DP problems, solutions to subproblems are reusable for larger problems. However, in Bin Packing, placing one item in a bin affects the availability of space for future items **non-linearly**.
    b. A small change (like moving one item) can cause a completely different packing structure, making it hard to break the problem into smaller reusable subproblems.
2. **Overlapping Subproblems Issue**:
    a. DP works best when subproblems recur frequently. But in Bin Packing, each arrangement of items in bins is unique and does not overlap significantly with others.
    b. Because of the combinational explosion of possible item arrangements (exponential in size), storing and reusing subproblem solutions becomes infeasible.

Additional Notes:

- Bin Packing is **NP-hard**, meaning there's **no known polynomial-time algorithm** that guarantees optimal solutions.
- Heuristics like **First-Fit**, **Best-Fit**, or **Next-Fit** are often used in practice instead of DP

Q3) (25 points) Develop a "toy" spelling checker application in Python using the bottom-up Levenshtein Edit Distance (LED) algorithm:

Program output:

Dictionary size: 29056

Please enter the file to check for spelling: misspelled.txt

- Suggestions for 'homan': [('woman', 1, 323), ('human', 1, 170), ('coman', 1, 17), ('roman', 1, 8), ('man', 2, 1648), ('women', 2, 385), ('home', 2, 294)]

- Suggestions for 'spote': [('spoke', 1, 218), ('spite', 1, 117), ('spot', 1, 76), ('spots', 1, 12), ('smote', 1, 4), ('spore', 1, 3), ('some', 2, 1536)]

- Suggestions for 'belst': [('best', 1, 268), ('beast', 1, 26), ('belt', 1, 11), ('felt', 2, 697), ('west', 2, 280), ('rest', 2, 206), ('else', 2, 201)]

- Suggestions for 'effrts': [('efforts', 1, 103), ('effort', 2, 130), ('effects', 2, 82), ('forts', 2, 8), ('exerts', 2, 3), ('effete', 2, 1), ('parts', 3, 295)]

- Suggestions for 'speling': [('spelling', 1, 4), ('feeling', 2, 362), ('seeing', 2, 207), ('speaking', 2, 185), ('swelling', 2, 164), ('smiling', 2, 161), ('opening', 2, 146)]

- Suggestions for 'perrfect': [('perfect', 1, 39), ('prefect', 2, 2), ('project', 3, 288), ('effect', 3, 187), ('perfectly', 3, 45), ('protect', 3, 41), ('correct', 3, 38)]

- Suggestions for 'avorage': [('average', 1, 18), ('voyage', 2, 12), ('forage', 2, 7), ('storage', 2, 3), ('averages', 2, 1), ('averaged', 2, 1), ('george', 3, 150)]

- Suggestions for 'typo': [('type', 1, 84), ('two', 2, 1137), ('too', 2, 548), ('top', 2, 42), ('types', 2, 33), ('tips', 2, 16), ('tap', 2, 10)]

- Suggestions for 'misspeling': [('missing', 3, 28), ('misspelled', 3, 1), ('listening', 4, 89), ('kissing', 4, 38), ('whispering', 4, 19), ('disputing', 4, 12), ('mingling', 4, 11)]

- Suggestions for 'mlch': [('much', 1, 671), ('such', 2, 1436), ('each', 2, 411), ('march', 2, 134), ('rich', 2, 92), ('match', 2, 41), ('mack', 2, 22)]

- Suggestions for 'mistkes': [('mistakes', 1, 15), ('mistaken', 2, 59), ('mistake', 2, 39), ('mistress', 2, 24), ('mister', 2, 2), ('misses', 2, 2), ('mists', 2, 1)]

- Suggestions for 'hauunt': [('haunt', 1, 2), ('aunt', 2, 52), ('hunt', 2, 31), ('gaunt', 2, 5), ('taunt', 2, 1), ('hand', 3, 834), ('count', 3, 748)]

- Suggestions for 'odoor': [('door', 1, 498), ('odour', 1, 19), ('odor', 1, 6), ('poor', 2, 128), ('floor', 2, 105), ('doors', 2, 47), ('doom', 2, 6)]

- Suggestions for 'colone': [('colonel', 1, 143), ('colony', 1, 57), ('colon', 1, 9), ('cologne', 1, 3), ('alone', 2, 337), ('close', 2, 219), ('colonies', 2, 202)]

Extra credit) (10 points)

This is for the BST and RB Balanced Tree implementation

Read the following postings:

- 1. https://stackoverflow.com/questions/2298165/pythons-standard-library-is-there-a-module-for-balanced-binary-tree

2. https://www.reddit.com/r/Python/comments/9allgh/wheres_pythons_damn_binary_search_tree
3. https://medium.com/@stephenagrice/how-to-implement-a-binary-search-tree-in-python-e1cdba29c533Links to an external site.
4. https://github.com/OmkarPathak/Data-Structures-using-Python/blob/master/Trees/BinarySearchTree.pyLinks to an external site.

Also, visit:

-
  - https://pypi.org/project/binarytree

I found the third posting above most useful for beginners in BST implementation. Please read the blog in its entirety.

1.
   1. Download or copy the BST implementation from the third or fourth link or any other source you like.
   2. Add the following methods to the downloaded code:
      - **getHeight** () - returns the height of a given tree
      - **countNodes**() - returns the number of total nodes in a given tree
   3. Create and then fill a BST tree with the integer data from 'rand1000000.txt' file and get the height and number of nodes of the tree using your methods above.
      - Make sure add() and delete() work properly
   4. **In-order traversal:** Build a new BST from "rand1000.txt" file (or 100 nodes if you prefer but you may need to make your own data file of 100 nodes)
      - Traverse the tree in ***descending*** order and print the keys with a ranking number. For example, if a key, 418621, is the 534th largest in the tree, display "[534] 418621"
      - You must use the Python "Generator" pattern using '***yield***' to display the number in the specified format.
   5. Now, do the same with an RB BST implementation. You may use the 'pypi' binary tree package (see the link below) or any other RB tree library you can find on the Internet. Then do steps 2 - 3 (you don't need step 4) with a Red-Black BST tree.
      - https://pypi.org/project/bintrees/ (a stopped project)
   6. Compare and analyze the height results from both trees (regular BST vs RB BST)

Binary Search Tree vs. Red-Black Tree Analysis

Binary Search Tree (BST):

- Inserted 1,000,000 values with no balancing mechanism

- Resulting tree height: 9,742

- Insertion time: ~21.47 seconds

- Became highly unbalanced (skewed due to insertion order)

Red-Black Tree (RB Tree):

- Simulated using a SortedSet for automatic balancing

- Inserted 1,000,000 values efficiently

- Estimated height: 20 (log2(N))

- Insertion time: ~2.18 seconds

Conclusion:

The RB Tree remained balanced and efficient even under large data loads. The regular BST, lacking balancing logic, grew very tall and inefficient for search and traversal. This experiment demonstrates the practical importance of self-balancing trees in real-world algorithms.