CS3000: Algorithms & Data — Summer 2025 — Laney Strange

Exam 2 Practice Problems

- These practice problems are to help you prepare for Exam 2. We'll release the solutions on Tuesday (June 10) so you can go over them and ask questions in your recitation.
- Exam 2 takes places during lecture on June 12, 2025.
- The exam will be due at the end of lecture (1:20pm). You'll hand it in on paper, but we'll scan it later for grading on gradescope.
- You may bring one 8.5x11-inch paper as a cheat sheet, with anything written or typed on it (one side only). You will submit this cheat sheet along with your exam, and you will not be permitted to use any other materials or notes during the exams.
- If you have a DAS accommodation for exams, please arrange to take the exam at their center. Make sure you schedule that time ASAP if you haven't yet!

Problem 1. *Practice-BST*

Recall that, in a binary search tree, the data in every node is larger than its left subtree and smaller than its right subtree. For the BST drawn above, for any node n, we have the following properties:

- *n.key* (the integer contained in the node)
- *n.left* (the node's left child; NIL if n is a leaf)
- *n.right* (the node's right child; NIL if n is a leaf)
- *n.p* (the node's parent; NIL if n is a root)

(a) Give complete pseudocode for a recursive post-order walk of a Binary Search Tree, i.e., recursively print the left and right subtrees before printing the root. Your function should take in one parameter, x, the root of the current subtree. You can assume you have access to a "print" function.

```
POST-WALK(x)
1  if x ≠ NIL
2       POST-WALK(x.left)
3       POST-WALK(x.right)
4       print x.key
```

(b) If you wanted to print out all the keys of a BST in sorted order, would you use a pre-order walk, in-order walk, or post-order walk?

Solution: in-order

(c) Suppose we're concerned with a node x in a BST and we want to know its successor in sorted order. If *x.right* is not NIL, meaning that it has a right subtree, give pseudocode that would return x's successor. (Hint: you can use any BST functions we've seen in class and/or homework.)

Solution: **return** TREE-MIN(*x.right*)

Problem 2. *Practice-BFS*

This is a practice problem. A similar problem in an exam situation would also provide the original BFS pseudocode.

If we know BFS will be working on a binary tree instead of a general unweighted graph, we can do it more simply.

Give pseudocode for a new version of BFS that, given a binary tree, computes and returns the height of the tree (i.e., number of "levels" in the tree). It should use a queue like BFS, but without coloring, and without any vertex attributes (like d or π).

Your function should take in the root node of the tree, T, and return the height of the tree, an integer. All nodes in your tree have a $n.left$ and a $n.right$ attribute, either/both of which would be NIL if the node doesn't have a child on that side. *Hint:* You can assume that your queue has a *length* attribute, which can help you keep track of all the nodes at your current level.

Solution:

```
BFS(T)
 1  Q = ∅
 2  ENQUEUE(Q, T)
 3  height = 0
 4  while Q ≠ ∅
 5       l = Q.length
 6       for i = 1 to l
 7            node = DEQUEUE(Q)
 8            if node.left ≠ NULL
 9                 ENQUEUE(Q, node.left)
10            if node.right ≠ NULL
11                 ENQUEUE(Q, node.right)
12       height = height + 1
13  return height
```

Problem 3. *Practice - Which Graph Algo*

For each problem on the left, give the best matching algorithm that could solve the problem from the right.

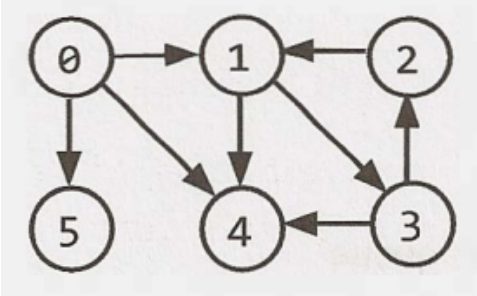
(1) The best way to lay out train tracks, if n cities need to have a station and you want to minimize the total amount of track used.	(a) Breadth-First Search
(2) An order in which you can take classes with pre-requisites.	(b) Topological Sort
(3) The fastest route to get from school to home.	(c) Minimum Spanning Tree
(4) How can an airline guarantee that they fly to all the places they say in their ads, but with as few flights as possible.	(d) Dijkstra's Algorithm
(5) The fewest number of clicks to get from one web page to another.	(e) APSP Algorithm
(6) The cheapest flight you can take, including possible stopovers, to get from Boston to Vancouver.	

Solution:

1 – c, 2 – b, 3 – d, 4 – c, 5 – a, 6 – d

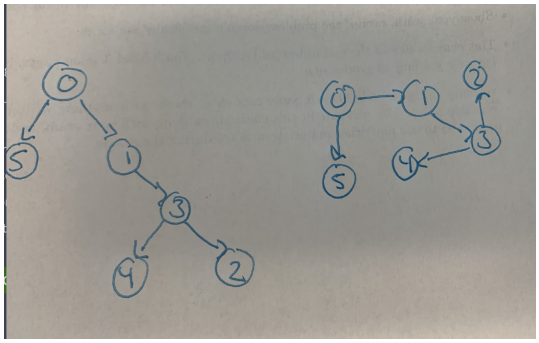
Problem 4. *Practice - DFS*

This problem is concerned with the directed graph below. In all subparts, assume that DFS is run on the graph starting with vertex 0, and that in case of a tie, the lower-valued vertex is visited first.



(a) Draw the DFS forest that would result after running the Depth-First Search algorithm on the graph.

Solution:



(b) Write out the vertices in the reverse-order in which they would be completed by the DFS algorithm.

Solution:
0, 5, 1, 3, 4, 2

(c) The answer you reached in the previous section is not a valid topological sort of the graph. Why is this the case?

Solution:
Topological sort can only be done on directed acyclic graphs. However, the above graph has a cycle, which creates a conflict: Node 1 has a dependency on 2, and Node 2 has a dependency on Node 3 which has a dependency on Node 1. It is impossible to meet all the constraints.

Problem 5. *Practice - Greedy*

Humpty Dumpty has fallen off the wall, and broken into n different pieces. All the king's horses are trying to put Humpty back together, but they need to make sure they have enough tape to repair him.

Humpty broke into n fragments of various sizes $s_1, s_2, \ldots, s_n$. Since bigger fragments are heavier, they need more tape to stick together. When we put two fragments with sizes s_i and s_j back together, we replace the two fragments with a single fragment of size $s_k = s_i + s_j$, and we use $s_i + s_j$ pieces of tape. We would like to completely put Humpty Dumpty back together, i.e. end with a single fragment, using the least amount of tape possible.

(a) Assume for this part that Humpty is size 28, and split into $n = 4$ pieces with sizes $[4, 5, 7, 12]$. What is the least amount of tape we can use to put Humpty together, and in what order do you connect the fragments?

Solution:

- Merge 4 and 5. Tape for this merge = 9, new fragment size = 9.
- Merge 7 and 9. Tape for this merge =16, new fragment size = 16.
- Merge 12 and 16. Tape for this merge = 28, new fragment size = 28.

In total we used tape worth $28 + 16 + 9 = 53$.

(b) For the general case, if you tackle this problem with a greedy approach, what would your greedy choice be at each step?

Solution: Always connect the two smallest fragments (just like Huffman!)

(c) Give pseudocode for your greedy strategy. Your algorithm will take as input the value $n > 0$ and a min-heap of positive integers $[s_1, ..., s_n]$, with each integer representing the size of a fragment. Your algorithm should return the value of an optimal solution (i.e., the total amount of tape used for a Humpty of size n).

Solution:

```
FixHumpty(S,n)
1  total = 0
2  while n > 1
3        u = ExtractMin(S)
4        v = ExtractMin(S)
5        Insert(S, u + v)
6        total = total + u + v
7        n = n − 1
8  return total
```

Problem 6. *Practice - DP*

The Fibonacci sequence follows a formula where the the nth Fibonacci number is the sum of the two previous numbers. There are two base cases: the first and second Fibbonacci numbers are both 1.

(a) Give pseudocode for a bottom-up Dynamic Programming algorithm that computes and returns the nth Fibonacci number.

Solution:

FIB(n)

```
1  let F[1..n] be a new array
2  F[1] = 1
3  F[2] = 1
4  for i = 3 to n
5        F[i] = F[i − 1] + F[i − 2]
6  return F[n]
```

(b) Show what your array/table looks like after calling your function with $n = 6$

Solution:$< 1, 1, 2, 3, 5, 8 >$

(c) Give a bound on the run-time of your algorithm

Solution:$\Theta(n)$

Problem 7. *Practice - Greedy*

You have an unlimited number of American coins with values: 50 cents, 25 cents, 10 cents, 5 cents, 1 cent. You want to determine the fewest number of coins needed to make some target.

(a) What's the fewest amount of coins you need to make 64 cents?

Solution:
$64 - 50 = 14$
$14 - 10 = 4$
$4 - 1 = 3$
$3 - 1 = 2$
$2 - 1 = 1$
$1 - 1 = 0$

6 coins are needed (one fifty-cent piece, one dime, four pennies).

(b) What's the fewest amount of coins you need to make 17 cents?

Solution:
$17 - 10 = 7$
$7 - 5 = 2$
$2 - 1 = 1$
$1 - 1 = 0$

4 coins are needed (one dime, one nickel, and two pennies).

(c) Write an algorithm that will find the fewest amount of coins needed to make n cents. You algorithm should take in n as well as c, an array of coin values with $c = [50, 25, 10, 5, 1]$, and cv, the length of array c. It returns the value of an optimal solution.

Solution:

MAKE-CHANGE(n, c, cv)

```
1  num_coins = 0
2  for i = 1 to cv
3       while n − c[i] ≥ 0
4            n = n − c[i]
5            num_coins = num_coins + 1
6  return num_coins
```

Problem 8. *MST- Kruskal*

This is a practice problem. On an exam, we would provide the pseudocode for reference.

You are given the below graph. Assume you want to make a minimum spanning tree using Kruskal's algorithm. Draw what the MST would look like (solid lines for edges in the tree, dotted for those in the graph that are not in the tree) and label the order the edges would be chosen according to the algorithm

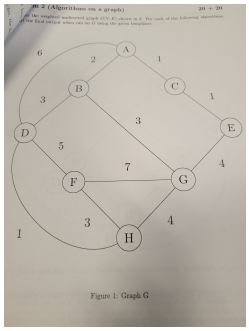
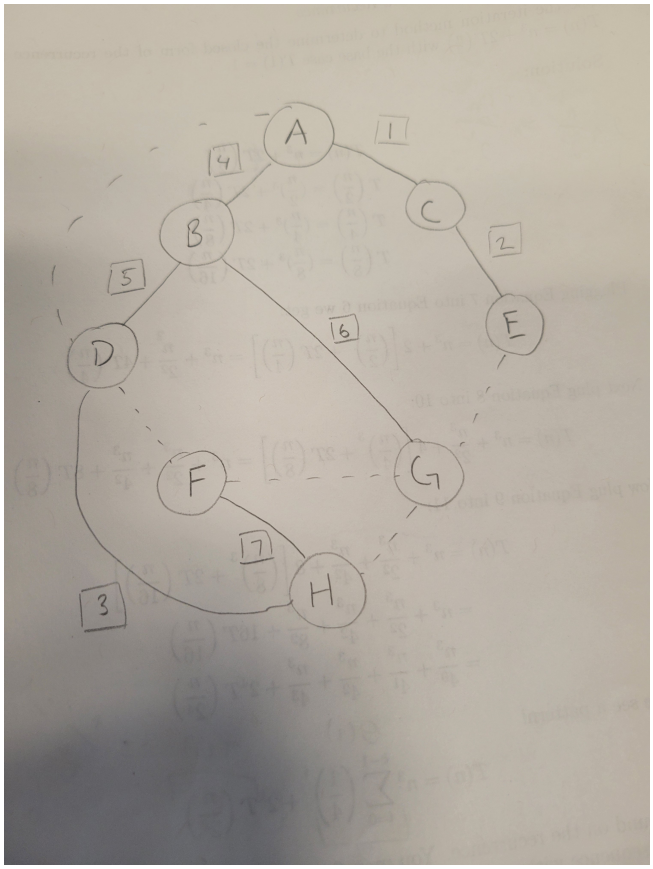


Figure 1: Graph G

Solution:

Note, some of the ordering of edges may be slightly different (only when breaking ties) but the MST should be the same.



Problem 9. *Practice - Binary Tree*

A binary tree of depth k with $n = 2^{k+1} - 1$ vertices is given. Every vertex contains a hidden real number, all numbers are different. The vertex is called *maximal* if its number is greater than the numbers in all adjacent vertices. You can use the following operation: choose any vertex and get its number. Give an algorithm which finds a maximal vertex using $O(k)$ such operations. Note that there could be several maximal vertices in the tree. You need to find only one of them, not all.

Solution:

Start in the root – check its number, it is our current number. On each step we are in the vertex, and number from this vertex is our current number. We check all adjacent vertices (which we have not checked yet). If all adjacent vertices has smaller numbers – our current vertex is a maximal vertex, we are done. Otherwise, move to any adjacent vertex with the number greater then the current number. The number from this vertex becomes new current number. Repeat steps until we found maximum. Since current number always increase, we will never return to already visited vertices. So our trajectory will be a non-self-intersecting path. The maximum length of non-self-intersecting path starting in the root of the tree of depth k is k. So we make at most k steps. Each step takes at most 3 operations. Thus the algorithm takes at most $3k$ operations.

Problem 10. *Practice - DP Shortest Paths*

Let $G(V, E)$ be a directed graph. Each edge $(u, v) \in E$ has an associated weight $w_{u,v}$ which could be positive, negative, or zero. Assuming G does not have a negative cycle (i.e., a cycle whose total weight is negative), we are interested in finding a path P from an origin node s to a destination node t with minimum total cost.

Because there are no negative cycles, you can assume that there is a simple path from s to t: if the minimum path takes n nodes, then you will use at most $n - 1$ edges.

Complete the recursive equation $OPT(i, v)$ below. Here, $OPT(i, v)$ represents the length of shortest path from a vertex v to destination t that uses $\leq i$ edges. (Feel free to write your answers below if they don't fit on the blank line.)

$$OPT(i,v) = \begin{cases} \infty & \text{if} \quad i = 0 \quad \text{and} \quad v \neq t \\ \_\_\_\_\_ & \text{if} \quad i = 0 \quad \text{and} \quad v = t \\ \_\_\_\_\_ & \text{if} \quad i > 0 \end{cases}$$

Solution:
Intuition: If we can't use any edges ($i = 0$) and the source node is the destination, then there is no cost! However, if we can't use any edges and the source node is not the destination, we can never reach it so we should put an infinity there to show it's impossible. Otherwise, we have two cases. Either:

- The shortest path from source to destination uses at most $i - 1$ edges. In this case, we can just look at the value from $OPT(i - 1, v)$
  OR
- The shortest path uses i edges. It will take one edge from source to some node w it can reach (with cost c_{vw}) and then $i - 1$ edges to get to from that w to t. We need to consider all valid $(v, w) \in E$ for this and we want to take whichever result gave us the minimum.

Once we've calculated both cases, we want to take the minimum of the two options as our answer.

$$OPT(i,v) = \begin{cases} \infty & \text{if } i = 0 \wedge v \neq \\ 0 & \text{if } i = 0 \wedge v = \\ min\{OPT(i-1,v), min_{(v,w)\in E} OPT(i-1,w) + c_{vw}\} & \text{if } i > 0 \end{cases}$$

Problem 11. *Practice - Graph Properties*

1. Show that every tree has at least one vertex with degree 1.
2. A bipartite graph is a graph G whose vertices can be divided into two sets, S and T, so that the only edges in G are edges between one vertex in S and one vertex in T. Give an algorithm to check if a graph is bipartite.
3. Show that a graph is bipartite if and only if it doesn't have a cycle of odd length (a cycle with 3 edges is an example of a cycle of odd length).

Solution:

1. **Proof by Contradiction**
   Assume by contradiction that all vertices in the tree has a degree of ≥ 2. Since it is a tree and has to be connected, there cannot be any vertex with degree 0.
   $\text{Sum}_{\text{degrees}} \geq 2|V|$.
   Each edge contributes 2 to $\text{Sum}_{\text{degrees}}$ since it is incident on 2 vertices.
   $\therefore$ number of edges in the graph $\geq |V|$. A tree with $|V|$ vertices can have atmost $|V| - 1$ edges, otherwise it causes cycle. Hence, we have a contradiction.
2. A DFS algorithm by assigning opposite colour to unvisited vertices and checking if adjacent visited vertex and the current vertex have the same colour.
3. Proof in both directions
   (a) If Graph G is bipartite, then G doesn't have a cycle of odd length.
   Let G($V = A \cup B, E$) is bipartite. Assume by contradiction that there is a cycle of odd length $v_1, v_2, ...v_k, v_1$ in G where k is odd. Let $v_1 \in A$. For the graph to be bipartite, v_i when i is odd must belong to A and when i is even must belong to B. Edge v_k, v_1 is an edge with both endpoints in A which is a contradiction.
   $\therefore$ If g is bipartite, then it cannot have a cycle of odd length.

   (b) if G is a graph with no cycle of odd length, then it is a bipartite graph.

   Let u $\in$ V and $A = \{u\} \cup \{v | d(u, v)$ is even$\}$ where $d(u, v)$ is the shortest path between the two vertices. Let B $= V - A$. Assume for contradiction that G is not bipartite and there exists an edge p, q with both p and q in A (or in B). Therefore $d(u, p)$ and $d(u, q)$ are both even (or both odd). The cycle given by $u \rightsquigarrow p \to q \rightsquigarrow u$ has length $d(u, p) + 1 + d(q, u)$ which is odd. This is a contradiction and hence there cannot be any such edge $\{p, q\}$ and G is bipartite.

   Alternate proof/Intuition for above proof: Start a BFS from a node u, color u white, vertices in the next level are colored black, the level after that are colored white and so on. Thus, all vertices at even levels $(0, 2, \dots)$ *are colored white and vertices at odd levels* $(1, 3, \dots)$ *are colored black. If there are no odd cycles, then there won't be edges between the same colored vertices.*

Problem 12. *Practice - Greedy Graph Algorithms*

Graph Coloring is an assignment of colors to vertices in a graph such that no two adjacent vertices share the same color. As an optimization problem, we try to color a graph with as few colors as possible.

(a) For the graph below, with vertices 1, 2, 3 and 4 sharing a color, and vertices 5, 6, 7, and 8 sharing a color, is the coloring an optimal solution?

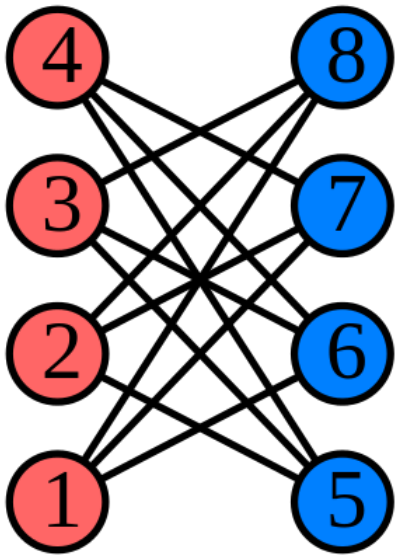


Figure 1: Vertices numbered 1, 2, 3, 4 are all colored Red while those numbered 5, 6, 7, 8 are all colored Blue.

Solution: Yes

(b) Suppose you attempt this optimization problem with a greedy approach: Your greedy algorithm takes in the graph G with adjacency-list representation and an array of colors c, with length $|G.V|$. Each vertex v would have an attribute $v.color$. Assuming the vertices are given in arbitrary order, visit each one of them and assign it the lowest-index color that is not assigned to a neighbor. Give a brief high-level description of how you would implement the algorithm. (Full pseudocode is not required here, but clear and descriptive bullet-points are.)

Solution:

One version of the algorithm would...

- For every vertex v, initialize $v.color = -1$
- Visit each vertex v
- Iterate over the color list
- For each color index i, visit each of v's neighbors u and determine whether $u.color == i$.

- If color index i not attached to any of v's neighbors becomes v's color.

(c) For your algorithm in Part B, what ordering of vertices would result in the coloring shown in Part A?

Solution: $1, 2, 3, 4, 5, 6, 7, 8$.

(d) Suppose you are given 8 colors, and the same graph above. However, you get very unlucky in the initial ordering of vertices, the worst possible ordering your algorithm could get. In this situation, how many colors would your greedy algorithm end up using?

Solution: 4.

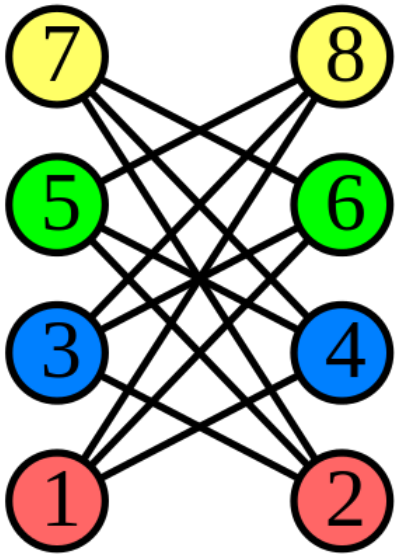


Figure 2: Example of vertex ordering which needs 4 colors to color the same bipartite graph.