CSC263 Winter 2021 Problem Set 3

Xiao Owen Hu & Eric Zhu

May 7, 2021

Question 1

a)

High level and data structure discussion:

Since each cell in the maze, e.g., the example maze, has possibly at least one direction, we may view this as a directed graph, where each cell is one node and each node has possibly more than one edge. A clearly sensible and relatively efficient solution would be to use BFS with some augmented adjacency list because it is possible for us to move in the maze with varying step sizes, d. So then, when we mutate d by landing on one of these red/yellow cells, we see that our possible paths are different than those with our previous d, i.e., we will consider different edges!

Given that this is inspired by Alice in Wonderland, we might view this as different ”floors” in Wonderland. We find this to be a wonderful analogy in that floors in a building may differ in layout, similar to how the different edges for each different d may lead to wildly different looking graphs, so we will use this analogy throughout question 1.

In algorithmic terms, we need an accompanying adjacency list for each d we encounter. So essentially, we'll just need to run BFS starting with our ”source” node, moving to the various appropriate adjacency lists for each d we encounter due to landing on either a red or yellow cell. We will use the same BFS algorithm described in the CLRS textbook, giving us an array of parents that we can later walk back through to the source and also an array of distances from the source. By keeping track of the walk, we may output an array that is exactly the shortest route Alice can take from the source to the goal. It will be comprised of the vertices (nodes) Alice would take rather than the edges. We can also find the length of the path by finding the appropriate entry for the goal in the distances from source array.

In the event that a solution can't be found, the path array will be empty and the distance value will be `inf`, i.e., `float("inf")`.

Algorithm:

Initialization

Given a text representation of the maze, we will first parse the text file to generate an adjacency list containing all the vertices as well as all possible edges. That is, for each vertex v in the adjacency list, we first calculate the corresponding one-indexed (x, y) coordinates for v, and once we have all the coordinates, we are able to calculate all of the possible edges for every possible d constrained to the grid size.

For example, suppose we have vertex v be the top left cell, i.e., v's coordinates are $(1, 1)$, and a 3 by 3 maze. Additionally suppose that v has only an arrow pointing right (east), then the adjacency list would be `[(2,1)]` when $d = 1$ and $[(3, 1)]$ when $d = 2$. Since each d has its corresponding own list for each v, we can store this information sensibly in a dictionary, providing us $\mathcal{O}(1)$ average case retrieval and insertion. More clearly, for our adjacency list, we provide each vertex with its own dictionary of possible edges, where the key is the step size d and the value is the list of possible edges corresponding to that d. Then we may add all of the vertices to an array to form our adjacency list, adj_lst. Note that for white cells (not to be confused with white coloured vertices during the BFS algorithm) we just do not add them to the adjacency list so no edge will be created with them.

As for implementation details, we will have a nested loop structure. In the outer loop, we loop over all of the vertices provided to us in the maze representation. In the inner loop, we construct all of the possible edges for a fixed vertex as described above. Specifically, we loop over all 8 directions So we can see that the worst case running time is $\mathcal{O}(n^2)$, where n is the number of cells in the maze. An example of the worst case is a node in the centre of the maze with arrows in all directions, which will lead to the inner loop having to construct an edge for every vertex in the maze, leading to a running time of $\mathcal{O}(n)$.

Modified BFS

Since we have the adjacency list, we will be able to execute our BFS algorithm. Our algorithm is largely based off of the algorithm provided in CLRS, with the additional feature of being able to handle changes in d.

Stemming from the multiple "floors" analogy above, we choose to implement many of our data structures here as dictionaries with their keys being their coordinate and their values being the corresponding list of values where the index minus 1 (to convert from one-indexing to zero-indexing) is their value a particular d. Note that these lists for each node has their size constrained to be the number of possible step sizes provided our `grid_width`, i.e., we cannot have more "floors" than the number of step sizes. This allows us to be consistent with our adjacency list, and allows us to handle the possible case where Alice returns to one of her visited cells but on a different "floor". We will refer to this as our **list replacement data structure**.

We choose dictionaries because in the worst case we know that dictionary accesses/insertions are $\mathcal{O}(n)$ and on average are $\mathcal{O}(1)$, so we won't increase the overall running time of our algorithm. Python additionally has hashing optimizations for its dictionaries so dictionaries were a very appealing choice.

We chose to implement the initialization portion of the BFS algorithm according to CLRS, i.e., we set each vertex's distance to `inf`, colour to white, and parent to `None`. Note that we chose to use our "list replacement data structure" for the array of the colours of each node, the array of distances from the source node, and the array of parents for each node. While doing so, we are able to extract the source node by checking the colour of the cell and seeing if it is "start", which is described more formally in part b.

With BFS initialization out of the way, we implement our BFS queue as an array because we are only allowed to import the `sys` library and all queue operations can be done in $\mathcal{O}(n)$ worst case using Python lists. Specifically, we implement `ENQUEUE(Q,v)` as `list.append(v)`, and we implement `DEQUEUE(Q)` as `list.pop(0)`. Here, we introduce another augmentation to the CLRS implementation of BFS. Since we have to consider d in order to get the correct list of edges for some given vertex and d value, we instead enqueue a tuple: `(d, v)`, where d is the d value after any possible increments/decrements and v is the corresponding node. Then, we enqueue the source node's tuple, set the distance for the source node to 0 for the corresponding initial d, and appropriately set the vertex colour to "grey" for the corresponding initial d.

We then loop over the queue as described in CLRS until it is either empty or we find the goal (which we check for at the end of the loop iteration). If we find the goal, we exit from the loop.

Note that we differ from the CLRS implementation in one respect, and in this paragraph we will refer variables using the variable names corresponding to the CLRS BFS implementation. Our difference is that we take into consideration the colour of the cell (node) and so we decrement/increment d appropriately for each v, which implies a few implementation differences. As d varies, we may have two different d values for some neighbour of u per iteration of the while loop, i.e., the original value corresponding to one "floor" and the modified value corresponding to the new "floor". So then, we must set the old "floor's" colour to "grey" by accessing the appropriate values contained in our colour "list replacement data structure" corresponding to the original d value. To update both the parent and distance data structures, we must be wary of d unlike the CLRS implementation as we need to set and get these values for and from the correct "floor". An example is updating the parent data structure for the top left red cell in the example maze. In that case, we would need to get the distance for u, node at (1,2), and use the value for d before it was incremented by 1. We would then set the distance for v, node at (1,1), using the incremented value for d.

Finally, if we exit the while loop because we have found the shortest path, we'll need to loop over the parents data structure to walk from the goal back to the source. Since we kept track of d for each v when we inserted v into the parents data structure, we know that we can get the correct path, and we can easily get the distance of the path by finding the entry in the distance data structure for our goal node.

b)

There are two sections in our representation of the maze.

The first section is the initialization of default values, in this case, the starting distance `travel_distance`, and the width of the maze `grid_width`.

The format of the initial section is one single header line and two lines for the declaration of the two values, `travel_distance=1` and `grid_width=3` for example. Note that an additional line, `grid_height`, may be added for rectangular mazes, but we are assuming square mazes for the sake of this question.

The second section contains the maze entries.

The format of the maze section is also one single header line and $\texttt{grid\_width}^2$ number of lines each representing a square in the maze.

The maze squares are ordered from left to right and top to bottom and displayed in lines from top to bottom. In other words, the first line under the maze header represents the top left square of the maze, and the second line represents the square to the right of the top left square, and so on and so forth.

Each line entry contains 8 bits indicating the directions of the arrows present in that particular maze square. Each bit, from the left-most to the right-most, represents north, south, west, east, northwest, northeast, southwest, and southeast, respectively.

Each line entry also contains the color or type of the arrow(s) in that particular maze square. The color is separated from the 8 bits using a vertical separator. There are additionally 3 colours of note: start, end, and white. The "start" colour indicates the "source" node, "end" indicates the goal (where Alice wants to go), and "white" indicates a cell Alice may not land on.

For example, a line entry representing a maze square with red arrows pointing towards the south, east and south east directions is as such:

```
0 1 0 1 0 0 0 1 | red
```

Whereas a line entry representing a starting square with arrows pointing towards the north and west would look like:

```
1 0 1 0 0 0 0 0 | start
```

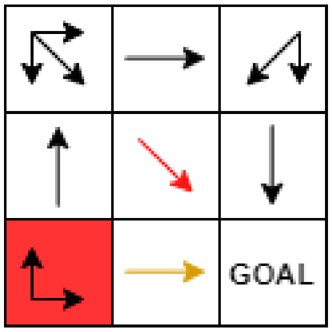
To demonstrate, the following is our representation of the small maze shown in the question.

```
-- initial --
travel_distance=1
grid_width=3
-- maze --
0 1 0 1 0 0 0 1 | red
0 0 0 0 0 0 0 0 | end
0 0 0 0 0 0 1 0 | yellow
1 0 0 0 0 0 0 0 | black
1 0 0 0 0 0 0 0 | black
0 0 0 0 0 0 1 0 | black
1 0 0 1 0 0 0 0 | start
0 0 0 1 0 0 0 0 | black
1 0 0 0 0 0 0 0 | black
```

c)

Completed and uploaded to Markus as `Alice.py`.

d)



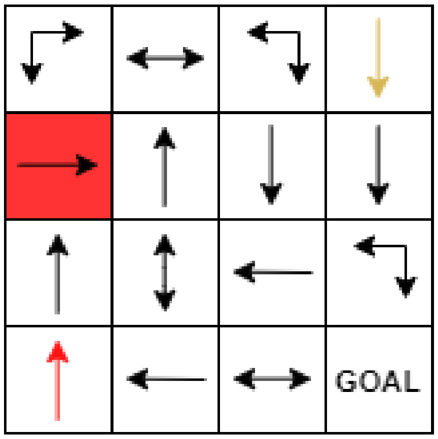
For our first test case (test1.txt), we will use a simple 3x3 maze that tests a variety of basic scenarios, namely diagonal traversal, stepping off the maze with increment, and terminating with a decrement to 0.

Here, we have a starting square on the bottom left that could seemingly find its way to the right and reach the goal. However, the yellow arrow should decrease the step distance to 0 which would cause the BFS traversal of that path to terminate.

The second closest path would've been the diagonal path from the top left to the bottom right through the center. However, the red arrow in the center should increment the step distance to 2 which would cause the path to step outside of the maze which is not a valid move.

Therefore, the closest and only path would reach around to the top left, top right and back down to the goal square while maintaining the starting step distance of 1.

And indeed, the output of our program shows a path length of 6 which follows the path starting at matrix position (1,3) to (1,2) to (1,1) to (2,1) to (3,1) to (3,2), ending at our goal (3,3). All indices are 1-indexed.



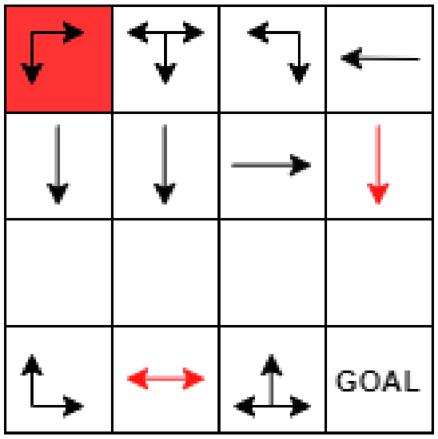
For our second test (test2.txt), we will test and demonstrate finding a path that requires both increasing and decreasing the step distance to reach the goal.

In this maze, we are required to arrive at the red arrow twice to increase the step distance to 3 in order to reach the yellow arrow that would eventually lead us to the goal after we decrease back down to 1.

This means that in our algorithm, we must not mark a node as visited if we have changed step distance as it is required in this case to arrive back at a "visited" red arrow as well as a "visited" yellow arrow.

In our program, we do discover this path that "cycles" back to the red arrow to increase the step distance once again from 2 to 3 which leads us to the top left and then the yellow arrow on the top right. Similarly, our program discovers the need to cycle around and revisit the yellow arrow to decrease the step distance back down to 1 in order for the final singular steps that lead to the goal.

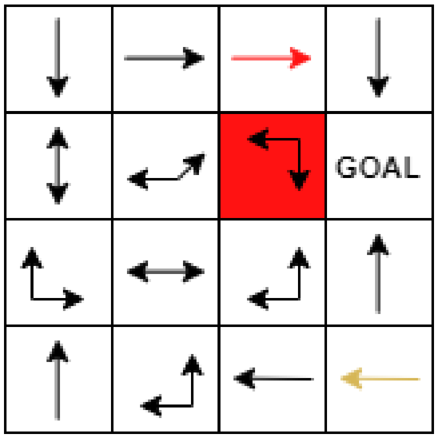
Our program outputs the correct path length 21 and the nodes that we visited in the order of traversal.



For our third test (test3.txt), we will introduce white squares to the maze which we must not step into. Our program deals with this by not adding an edge to the adjacency list if it ends with a white square.

In this maze, because of the white squares on the third row, we cannot step into the red arrow on the bottom row early since crossing the white row requires a step distance of 2. Therefore, the only way of reaching the end goal is by arriving at the red arrow on the top half of the maze and crossing the white row.

This is verified by our program output which prints the path going from the top left, turning at the third square of the first row, turning again at the third square of the second row, and finally arriving at the red arrow that allows us to cross over to the goal. It also correctly outputs the path length 5.



For our last test (test4.txt), we will test our program's behavior with an unsolvable maze.

All BFS traversals for this graph would terminate for a step distance of 1 as nodes become visited and a cycle is formed, and the red arrow causes us to step outside of the maze boundary which will also terminate the program.

Indeed, our program prints out ”No solution found” along with an empty path array and a path distance of infinity.

Question 2

(a)

Given the adjacency list of an undirected graph with $2n$ nodes, we know that there are n nodes of degree one and n nodes of degree three in the cycle.

Main idea: start with a cycle node and mark it as the starting point, perform a modified DFS (while making sure the properties of a sun are not violated) until we find an edge that leads back to the starting node, and confirm that we have navigated the entire graph.

Algorithm pseudocode:

```
IS_SUN(G=(V,E)):
    # initialization
    for each v in V:
        colour[v] = white
    # global variables
    leaf_count = 0
    not_a_sun = false
    vertex_end = null
    vertex_start = FIND_VERTEX_START(G)
    if vertex_start == null:
        return false
    # DFS search
    DFS-VISIT(G, vertex_start)
    # final check
    if leaf_count == n and vertex_end != null and !not_a_sun:
        return true
    else
        return false

FIND_VERTEX_START(G=(V,E)):
    if number of (V[0],v) in E == 3:
        return V[0]
    else if number of (V[0], v) in E == 1:
        # check its only neighbor
        u = neighbor of V[0]
        if number of (u,v) in E == 3:
            return u
        else:
            return null
    else:
        return null

DFS-VISIT(G=(V,E),u):
    colour[u] = grey
    # validate neighbor count
    neighbor_count = number of (u,v) in E
    local_leaf_count = 0
```

```
        if neighbor_count == 1:
            leaf_count++
        else if neighbor_count != 3:
            not_a_sun = true
        # examine edges
        for each (u,v) in E:
            if not_a_sun:
                return 0
            if v == vertex_start:
                # completed cycle
                vertex_end = u
            if colour[v] == white:
                local_leaf_count += DFS-VISIT(G,v)
        colour[u] = black
        # validate leaf count
        if local_leaf_count != 1 and neighbor_count != 1:
            not_a_sun = true
        # return 1 if current vertex is a leaf
        if neighbor_count == 1:
            return 1
        else:
            return 0
```

Explanation:

We first perform an initialization that marks every vertex white and define three global variables:

1. Integer variable that counts the number of nodes of degree one (we will call them leaves) that we have traversed.

2. Boolean variable that indicates that the graph is not a sun and should terminate early

3. Variable that stores the vertex that leads back to the starting vertex

Then we do an initial search for the first cycle node by going to the first vertex on the list and checking the number of neighbors:

- If it contains exactly 3 neighbors, we mark that vertex as the starting node

- If it contains exactly 1, we traverse to its only neighbor and check if that vertex has exactly three neighbors: if it does, mark that vertex as the starting node; otherwise, we can conclude that the graph is not a sun.

- If it contains neither 1 nor 3 neighbors, we can conclude that the graph is not a sun

Now that we have found our starting vertex, we are ready to do DFS with that as our starting point.

Instead of looping through each vertex and performing DFS-VISIT given they are still marked white, we will do a single recursive DFS-VISIT call to the starting vertex because a sun should be a single connected graph.

DFS-VISIT:

For each DFS-VISIT call, we first colour the current vertex grey, initialize a local leaf count variable, and examine its neighborhood. For the graph to be a valid sun, each vertex must contain either 1 or 3 neighbors. If it contains exactly 1 neighbor, then we increment the leaf count; if it contains neither 1 nor 3 neighbors, then we set not_a_sun to true. (Line 34-38)

Once we have validated the vertex, we will loop through its neighbors and examine them. (Line 43-50)

First, if not_a_sun has been set to true, the function will return early. We have this check inside the for loop so that any future DFS-VISIT that sets not_a_sun to true will cause each recursion backtrack to return early as well.

We also check if one of the neighbors is the starting vertex and set vertex_end to the current vertex if it is. Lastly, if the neighbor vertex has not been visited (colour white), we call a recursive DFS-VISIT on that vertex and increment local leaf count by its return value.

After we have visited and recursed through every neighbor, we colour the current vertex black and validate that we have exactly one neighbor leaf (given the current vertex is not a leaf itself), setting not_a_sun to true if it does not. (Line 51-54)

Finally, we will return 1 if the current vertex is a leaf and 0 otherwise. (Line 56-59)

Back to main program:

After the main DFS-VISIT call returns, the program could be in two states:

- not_a_sun was set to true in which case the main function returns false.

- Every vertex connected to the starting vertex is traversed and meets the vertex requirements for a sun.

For the latter scenario, we then need to check if the leaf count equals n and a cycle back to the starting vertex is found.

If both conditions are satisfied and not_a_sun is false, then we conclude that the graph is a sun and return true.

b)

Since we know that there is a single cycle of length n in the graph, there must be at least n vertices in the cycle. And since each vertex in the cycle is connected to exactly 1 node of degree one, there must be at least n vertices of degree one. Because there are a total of $2n$ vertices, there must be exactly n vertices in the cycle and n connected vertices of degree one.

Additionally, since each vertex in the cycle contains 2 neighbors that are also vertices part of the cycle and exactly 1 neighbor of degree one, they must have degree 3.

In our algorithm, we perform a single DFS starting from a vertex in the cycle. We examine every connected vertex and make sure that they are either a vertex of degree one or vertex of degree 3 and that each vertex of degree 3 has exactly one neighbor of degree one. If any of these properties fails, the program will return false.

After we have examined every connected vertex, each vertex must have satisfied the above requirements for the program to return true. Then, we will verify that we have discovered the vertex that would complete the cycle during our DFS. Lastly, we check if the total number of nodes of degree one that we have discovered is n. If all these conditions are satisfied, we return true.

Therefore, the only way for our algorithm to return true is if we have visited exactly n vertices of degree one and every other visited vertex has degree three. Since the n vertices have degree one, they each must be connected to exactly one vertex. And because we have verified that each vertex of degree three is connected to exactly one node of degree one, there must be n vertices of degree 3 that are connected. Finally, because we have verified that a vertex that connects back to the starting vertex has been discovered, we can conclude that a cycle exists and the graph is a sun.

(c)

For our initialization, we loop through the list of vertices so it will cost $2n = \Theta(n)$ time.

For finding our starting vertex, we examine the neighbors of the first vertex and possibly the neighbors of its neighbor. For each of the two vertices, we stop if we find out that there are more than 3 neighbors, so its worst-case runtime is $2 \times 3 = 6 = \Theta(1)$.

For the main DFS search, we traverse through all the vertices that are connected to the starting vertex at most once each because we mark every visited vertex. In the worst case, all vertices are connected and we would visit $2n$ vertices. For each DFS-VISIT of those $2n$ vertices, we look through their adjacency list and stop if there are more than 3 neighbors.

Therefore, we examine at most 3 neighbors for each of the $2n$ vertices, so the worst-case runtime is $2n \times 3 = 6n = \Theta(n)$.

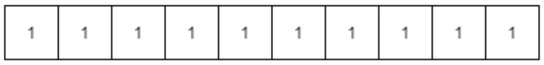
The final check of the main program is in constant time, so adding together the previous three sections, the total runtime of the program is $O(n)$.

Question 3

(a)

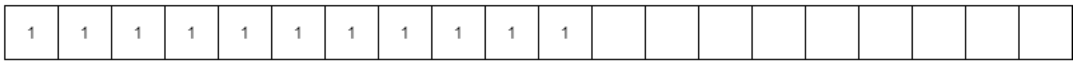
This data structure runs out of money after the 41st INSERT operation. Figure on next page.

After first 10 INSERT operations



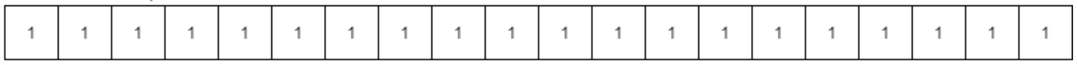
Credits: $40

After the 11th INSERT:



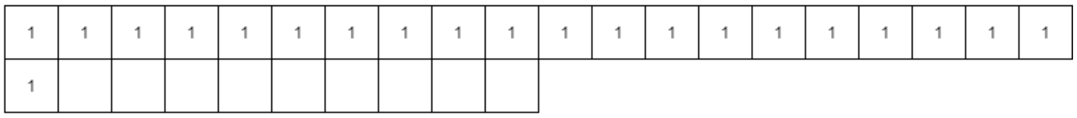
Cost: $21, Charge: $5, Credits: 40+5-21=$24

After 20 INSERT operations:



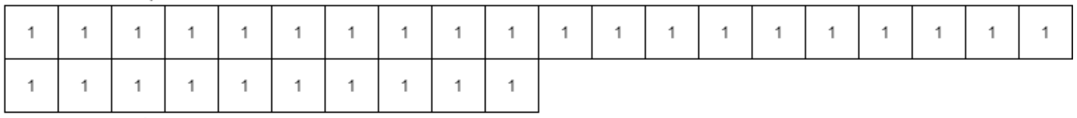
Credits: 24+9*4=$60

After the 21th INSERT:



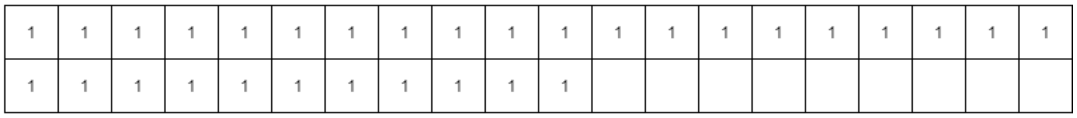
Cost: $41, Charge: $5, Credits: 60+5-41=$24

After 30 INSERT operations:



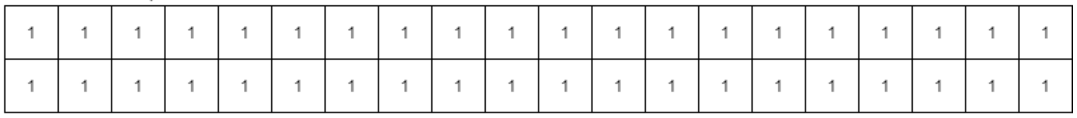
Credits: 24+9*4=$60

After the 31th INSERT:



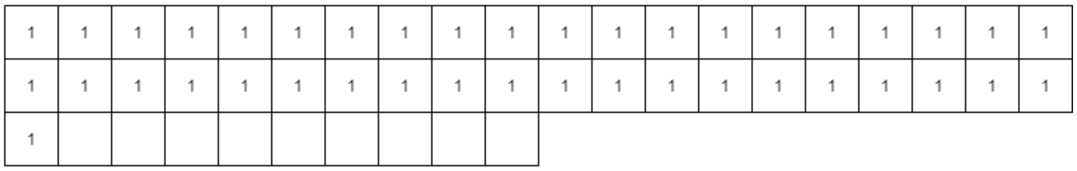
Cost: $61, Charge: $5, Credits: 60+5-61=$4

After 40 INSERT operations:



Credits: 4+9*4=$40

After the 41st INSERT:



Cost: $81, Charge: $5, Credits: 40+5-81=-$36

(b)

We will prove that it never runs out of money.

Proof by induction:

Let $b \in \mathbb{Z}^+$ be the initial capacity and the capacity increment.

Assume that the charge starts with \$3 per INSERT and increases by \$2 after each multiple of b calls to INSERT.

This means every INSERT call between the $(kb+1)$-th and the $(k+1)b$-th (inclusive) will be charged $\$3 + 2k$ where $k \in \mathbb{N}$.

Assume that a non-array-growing INSERT costs \$1 and every $(kb+1)$-th INSERT call costs $\$2kb + 1$ (similar to the lecture notes for dynamic array).

Credit Invariant: When the array needs to grow, the current credits are enough to cover the difference between the charge and the cost of the array-growing INSERT call.

In other words, define predicate $P(n)$: the amount of credits left after $nb + 1$ INSERT operations is greater or equal to 0.

WTP: $\forall n \in \mathbb{Z}^+$, $P(n)$

Justification for this being sufficient: we are only concerned with the $(nb + 1)$-th INSERT operations (where n is a positive integer) because those have the expensive costs that require growing the array, whereas all other INSERT operations cost \$1 but are charged $3 + 2k$ so the money would never run out during those operations. Therefore, if we prove that the remaining credits after each $(nb + 1)$-th INSERT operation are greater or equal to 0, then we have effectively proven that the credits never run out.

Base case: $n = 1$, WTS: $P(1)$

After the first b INSERT operations, the array will be filled and we will have $(3 - 1)b = 2b$ credits left (3 was the charge and 1 was the actual cost).

When we execute the $(b + 1)$-th INSERT operation which is an array-growing operation, the charge is $3 + 2k = 5$ when the actual cost is $2kb + 1 = 2b + 1$.

Since we had $2b$ credits after the first b INSERT operations,

$$\begin{aligned}\text{the amount of credits left} &= \text{previous credits} + \text{charge of INSERT} - \text{cost of INSERT}\\ &= 2b + 5 - (2b+1)\\ &= 4 \geq 0 \implies P(1)\end{aligned}$$

Induction step: Assume $P(n)$, WTS: $P(n+1)$

From the induction hypothesis, we assume $P(n)$ holds. Therefore, after $nb+1$ INSERT operations, we have 0 or more credits left.

Since we know that the $(kb+1)$-th INSERT grows the array by b, the next array-growing operation will be the $(nb+1)+b = [(n+1)b+1]$-th INSERT. Therefore, every INSERT between the $(nb+2)$-th and the $(n+1)b$-th (inclusive) is non-array-growing, costs \$1 each and is charged $3+2n$.

This means that we have $b-1$ operations that cost \$1 each and are charged $3+2n$. So, after $(n+1)b$ operations, we will have $(3+2n-1)(b-1) = 2(n+1)(b-1)$ extra credits.

When we execute the $[(n+1)b+1]$-th INSERT operation, the array needs to grow and the charge becomes $3+2k = 3+2(n+1)$ and the cost is $2kb+1 = 2(n+1)b+1$.

Therefore, after the $[(n+1)b+1]$-th INSERT operation,

$$\begin{aligned}\text{The amount of credits left} &= \text{previous credits} + \text{charge of INSERT} - \text{cost of INSERT}\\ &= 2(n+1)(b-1) + [3+2(n+1)] - [2(n+1)b+1]\\ &= 2(n+1)b + 3 - 2(n+1)b - 1\\ &= 2 \geq 0 \implies P(n+1)\end{aligned}$$

■

c)

We begin our aggregate analysis by splitting up our problem. From the question, we know that we have $m \in \mathbb{N}$ `INSERT` operations, starting with an empty array. We are additionally given $b \in \mathbb{N}$, which is both the initial capacity and the capacity increment. So then, like the dynamic array example from class, we know that the costly non-constant time `INSERT` operations come from the dynamic array expansion (due to a full array). As such, we will aim to find both the exact cost of each array expansion and the times the array expands in m `INSERT` operations so that we are able to accurately find the exact cost of m `INSERT` operations for a given b.

First, suppose we have call $i \in \mathbb{N}, i \leq m$ (the i^{th} call), where call i causes the array to expand. For example, if $b = 10, m = 20$ and if $i = 11$, we would see an expansion. At call i, we would need to perform a few tasks as seen in week 7 lectures:

1. Allocate space for all $i - 1$ elements
2. Insert all $i - 1$ elements in new array
3. Insert new element

Tasks 1 and 2 would take the same amount of time since they would operate on the same number of elements ($i - 1$ elements), and allocating space for an elements/inserting a value into a memory address are both constant time operations that cost 1 operation each. Since we know at call i we have exactly $i - 1$ elements total (one element inserted per call), we know that there would be exactly $2(i - 1)$ operations to both allocate space for $i - 1$ elements and insert all $i - 1$ elements into the new array. Then, it would take exactly 1 operation to insert the new element. So the total cost for call i would be exactly $2i - 2 + 1 = 2i - 1$.

On the other hand, when `INSERT` does not cause an array expansion, we find that the total cost is simply 1: the cost it takes to insert the element into memory. Therefore, we can form the set of equations to describe the cost of `INSERT` for the k^{th} call in our sequence of m calls:

$$t_k = \begin{cases} 2k - 1 \text{ if an array expansion is needed} \\ 1 \text{ otherwise} \end{cases}$$

Our last missing piece to this solving this problem is finding the number of array expansions. So recall that we have m total `INSERT` operations with an initial capacity and capacity increment of b. Since our initial capacity is b, we would first need b insert calls in order to fill up our original array, then we would need to increase the capacity of the array on the next call, i.e., $b + 1$ calls. With this in mind, we can come up with the exact number of array expansions. So we increment by a constant amount, b, we know that it will always take exactly b insertions after an array expansion to cause another array expansion. Thus, we know that $\lceil \frac{m}{b} \rceil$ accounts all currently filled arrays and the possibly partially filled array caused by the most recent array expansion. In other words, the $\lceil \frac{m}{b} \rceil$ filled arrays account for every array expansion as we must have an array expansion to have a filled array with one big caveat. The caveat is $\lceil \frac{m}{b} \rceil$ over counts the number of expansions because as noted above, our first array expansion takes $b + 1$ calls due to being given an initial size of b, so we must do $\lceil \frac{m}{b} \rceil - 1$ to find the number of array expansions. Clearly, we'd get a negative number of we have $m = 0$, but in that case, we just note that the cost is 0.

Now we can create a equation to find the total cost of m operations, $T(m)$:

$$\begin{aligned}
T(m) &= \sum_{k=1}^{m} t_k \\
&= \sum_{i=1}^{\lceil \frac{m}{b} \rceil - 1} 2 \cdot b \cdot i - 1 + \sum_{i=\lceil \frac{m}{b} \rceil}^{m} 1 && \text{(Split the } m \text{ operations according to cost)} \\
&= 2b \sum_{i=1}^{\lceil \frac{m}{b} \rceil - 1} i - \sum_{i=1}^{\lceil \frac{m}{b} \rceil - 1} 1 + \sum_{i=\lceil \frac{m}{b} \rceil}^{m} 1 \\
&= 2b \cdot \frac{(\lceil \frac{m}{b} \rceil - 1)(\lceil \frac{m}{b} \rceil)}{2} - \sum_{i=1}^{\lceil \frac{m}{b} \rceil - 1} 1 + \sum_{i=\lceil \frac{m}{b} \rceil}^{m} 1 \\
&= b \cdot (\left\lceil \frac{m}{b} \right\rceil - 1)(\left\lceil \frac{m}{b} \right\rceil) - (\left\lceil \frac{m}{b} \right\rceil - 1) + (m - \left\lceil \frac{m}{b} \right\rceil)
\end{aligned}$$

Note that when we split the m operations according to cost above, we transform the bounds of our summations into the counts for the respective costs.

Then we see from above that since our operation count is in terms of m, the last terms of the equation above, i.e., $(\lceil \frac{m}{b} \rceil - 1), (m - \lceil \frac{m}{b} \rceil)$ are $\Theta(m)$. So we rewrite the above equation as:

$$b \cdot (\left\lceil \frac{m}{b} \right\rceil - 1)(\left\lceil \frac{m}{b} \right\rceil) - \Theta(m) + \Theta(m)$$

Then we'll examine the first term, i.e., $b \cdot (\lceil \frac{m}{b} \rceil - 1)(\lceil \frac{m}{b} \rceil)$. We will rewrite it as:

$$(b \left\lceil \frac{m}{b} \right\rceil \left\lceil \frac{m}{b} \right\rceil - b \left\lceil \frac{m}{b} \right\rceil)$$

We can clearly see that $b \lceil \frac{m}{b} \rceil \lceil \frac{m}{b} \rceil$ is a higher order term than $\lceil \frac{m}{b} \rceil b$, so we will only consider $b \lceil \frac{m}{b} \rceil \lceil \frac{m}{b} \rceil$ as we wish to obtain a big-Theta bound.

By the definition of ceiling from CSC165 we know that $\frac{m}{b} \leq \lceil \frac{m}{b} \rceil < \frac{m}{b} + 1$. Therefore, we can bound $b \lceil \frac{m}{b} \rceil \lceil \frac{m}{b} \rceil$ using the definition of floor:

$$b \left\lceil \frac{m}{b} \right\rceil \left\lceil \frac{m}{b} \right\rceil \iff b(\frac{m}{b})^2 \leq b \left\lceil \frac{m}{b} \right\rceil \left\lceil \frac{m}{b} \right\rceil < b(\frac{m}{b} + 1)^2$$

$$b(\frac{m}{b})^2 \leq b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil < b(\frac{m}{b}+1)^2$$
$$\iff$$
$$\frac{m^2}{b} \leq b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil < b((\frac{m}{b})^2 + 2\frac{m}{b} + 1)$$
$$\iff$$
$$\frac{m^2}{b} \leq b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil < \frac{m^2}{b} + 2m + b$$

Thus we see that in terms of big theta, we have that both of the bounds of $b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil$ are in $\Theta(m^2)$ as $\frac{m^2}{b} \in \Theta(m^2)$ and $\frac{m^2}{b} + 2m + b \in \Theta(m^2)$. Therefore, we can say that $b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil - b\left\lceil\frac{m}{b}\right\rceil \in \Theta(m^2)$ as we noted above that $b\left\lceil\frac{m}{b}\right\rceil$ is a lower order term. So then plugging in our $\Theta(m^2)$ bound for $b\left\lceil\frac{m}{b}\right\rceil\left\lceil\frac{m}{b}\right\rceil - b\left\lceil\frac{m}{b}\right\rceil$, we obtain this expression for $T(m)$ using our partial big-Theta expression above:

$$T(m) \in \Theta(m^2) - \Theta(m) + \Theta(m)$$

Clearly $T(m) \in \Theta(m^2)$, and since we have a sequence of m `INSERT` operations, we have as our amortized complexity for `INSERT` to be $\Theta(m)$.

Question 4

a)

The key observation is that path compression will shorten the height of a tree, and thus, change the structures. So our goal is to find the smallest number of operations that results in a tree that will makes $T1, T2$ different after path compression.

Recall that path compression aims to reduce traversal time by replacing the parent pointer of each node along a path with the representative of the disjoint set. So in that vein, we'd want to construct a tree with a branch with more than 1 node in the least amount of operations as possible.

Since $T1, T2$ both use union by weight, it would be sensible for us to find consider trees of the same weight since that would cause branches of different lengths. Constructing two trees each with a weight of 1 doesn't help us; path compression wouldn't change the tree representation. So we consider trees each with a weight of 2.

When we have trees each with a weight of 2, we can union them to obtain a tree with a representative, a branch with 1 node, and another branch with 2 nodes. It then follows that if we were to call `FIND-SET` on the deepest node in the branch with 2 nodes, we would have differing $T1, T2$ due to path compression as that node would have its parent replaced with the representative.

So we will do exactly that. First, we construct 2 trees of weight 2: `MAKE-SET(1)`, `MAKE-SET(2)`, `UNION(1,2)` and `MAKE-SET(3)`, `MAKE-SET(4)`, `UNION(3,4)`. Then we union these two trees: `UNION(1,3)`. And finally, we call `FIND-SET(4)`, which performs the path compression, replacing the parent pointer of that node with the representative of the disjoint set, i.e., the node containing 1.

In this process, we performed 8 operations, so we find $n = 8$, justified by our analysis above.

b)

The key observation is precisely the difference between union by weight and union by rank; that is, the rank of a disjoint set tree is a monotonically increasing function while the weight of a disjoint set tree is a strictly increasing function. So to find n, the smallest number of operations to make $T2, T3$ differ, we find the smallest sequence of operations that increase weight but not rank. Also note that we must also change the structure between the trees, such as the representative.

Recall that the rank of a tree increases if and only if on a `UNION` operation, the ranks of the two trees are exactly the same. So then, since the weight of a tree is strictly increasing as we add more and more nodes to the tree, we see that our analysis of n is solely dependent on constructing trees with more than 1 node and same rank then calling the `UNION` operation on them, so that we get $T2$ to have a tree with a larger weight than that of $T3$'s rank.

We begin with two `MAKE-SET` calls and one `UNION` call: `MAKE-SET(1), MAKE-SET(2), UNION(1,2)`. We see that $T2$ now has a tree of weight 2 (2 nodes), and $T3$ has a tree of rank 1.

Next, to potentially change either the rank of $T3$'s tree or the weight of $T2$'s tree, we must create more nodes and combine these disjoint sets, i.e., more `MAKE-SET` and `UNION` calls.

So let us examine what happens with one more `MAKE-SET` and `UNION` calls. Suppose we do `MAKE-SET(4), UNION(1,4)` in that order. We would see that $T2$ now has a tree of weight 3 because of the additional node, while $T3$ still has a tree of rank 1. This follows from our key observation in that we may have a tree with 2 nodes that has a rank of 1 but also a tree with 3 nodes that has a rank of 1. However, these two trees would have two different weights.

At this point we could continually call `MAKE-SET` and `UNION` between our tree with 3 nodes and one new node but that would preserve the same tree structure between $T2, T3$ because both the weight and rank of $T2, T3$ respectively are bigger than those of any single node (disjoint set). So then, we would need to union our current tree of $T2, T3$ with a new tree that has more than 1 node. From the previous paragraph, we noted that while we added an additional node, our rank didn't increase for $T3$ while the weight for $T2$ did. That means, we could do two more calls to `MAKE-SET` and union those two single element disjoint sets to create a new tree with rank 1 and weight 2, which would allow us to pick a new representative in $T3$ (either the representative of the tree with 2 nodes or our current one) through a union call.

We'll do exactly that. So suppose we perform 2 more `MAKE-SET` calls and more `UNION` call: `MAKE-SET(5), MAKE-SET(6), UNION(5,6)`. Then we will choose to perform `UNION(5, 1)`. We do this to choose the representative during the union in $T3$ to be the representative of the smaller tree, i.e., the one containing only 5 and 6. We can do this because 5 is the first argument to the `UNION` call. Interestingly, this union call doesn't affect the order in $T2$ since one of the trees had a weight of 3 and the other tree with 2 nodes only had a weight of 2. Thus, for $T2$, the representative after the union call didn't change, and $T2, T3$ now have trees with different representatives.

Counting up our sequence, we find that we performed 9 operations, which we conclude to be n, i.e., $n = 9$, justified by our analysis above.