Kuja/Jecht Problem Set Solution Sketches

Name: ALCALA, Christian
CANSANA, Jose Lorenzo
PE, Sean Michael
Section: CSALGCM S15

	Poor (0.0)	Satisfactory (2.0)	Good (3.0)	Excellent (4.0)
Depth of Discussion (Rx10)	Solution is unclear or non-existent	Solution is slightly explained, but on a very top-level or shallow level of understanding. **Recurrence is given.**	Solution/thought process is explained well, but there are aspects that are missing or assumptions that are not explained. **Recurrence is given. Examples of overlapping subproblems are illustrated using a figure or example.**	Solution/thought process is explained thoroughly and the logical line of thinking is clear and concise. The line of thought of the author is easy to follow. **Recurrence is given. Examples of overlapping subproblems are illustrated using a figure or example. For optimization problems, optimal substructures are shown.**
Correctness (Rx3)	Solution made no attempt to produce correct input or to run within the appropriate time limit.	Solution is far from a correct solution or is a naive approach that is inefficient. **Recurrence has correct state but most of the values/transitions are incorrect.**	Solution is correct or almost correct, but there are holes or aspects that are questionable or unclear. **Recurrence has correct state but a few of the values/transitions are incorrect.**	Solution is correct and is explained thoroughly and well. **Recurrence has correct state and all of the values/transitions are correct.**
Writing (Rx2)	Writing is full of grammatical and typographical errors. Structure of	There are some grammatical and typographical errors. The writeup has some semblance of structure, but it is	Very few grammatical and typographical errors. Write-up is structured well.	No grammatical and typographical errors. Write-up is exceptionally structured i.e. the flow of discussion is intuitive and smooth..

	write up is awkward and disorganized.	confusing to read.		

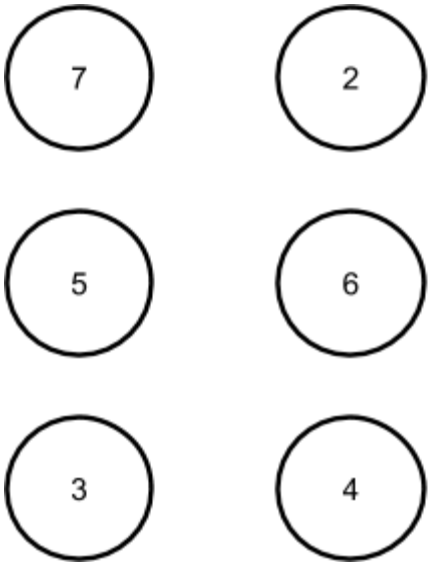
Problem Kuja-C: Which Line Is It Anyway? (Medium Version)
Status: Accepted
Difficulty Rating: 3
Solution Sketch/Write-Up/Narrative:

In this problem, the program is to find the minimum total distance from the start row to the last row of a matrix. The distance is computed by adding the weights of each node traversed in the matrix. Though, in this problem, everytime you switch to a different column, k value is added. The user will input 4 things: an nxm matrix size where n and m are integers, a k integer value that will be added to column switches, and the contents of the n x m matrix. The program can start at any column at the first row and can end at any column at the last row as long as it satisfies the minimum total distance required.

To find the minimum total distance, the program must try all possible minimum total distances from different start points and end points. This is a comparison that is done at the end of every iteration of traversal. Since the problem exists in a 2D matrix, and the goal is to get to the last row of the matrix, the possible moves that could be made will be traversing to the node on the left column of the current column in the next row (row-1, column-1), to the node in the same column in the next row, (row-1, column), and to the node on the right column of the current column in the next row (row-1, column+1). This will serve as the transitions of our Dynamic Programming Algorithm. Though, due to restrictions of the matrix, we must consider the possible states that will exist in the algorithm. There will be a case when the pointer is at the left-most column, therefore the pointer can only go down the same column or move to the right column on the next row. Another case will be when the pointer is at the right-most column where the pointer will be restricted to going down the same column or move to the left column on the next row. The general case will be when the pointer is in neither extreme columns or in the middle, the pointer can do to all 3 possible moves. These cases contain a self-method call (if bottom-up approach is used) due to the recurring nature of Dynamic Program. Also, all cases returns the minimum distance of their self-method calls. These returned values are added to the total distance traveled. These returned values are then saved to the memo table for future reference. To terminate this, a base case must be reached. The base case is reached when the row value is 0 or in other words, the pointer reached the end row. The base case just returns the current node weight to be added to the total distance traveled. After which, the comparison is done with the previous minimum total distance and the current traversal's total distance -- the minimum is then stored and returned after all possible combinations are made. With these, the recurrence function is:

$$f(row,\ col) = \begin{cases} time[row][col] & row = 0 \\ min(k + f(row-1, col-1), & row > 0 \wedge col > 0 \\ f(row-1, col), k + f(row-1, col+1)) + \\ time[row][col] & \\ min(f(row-1, col), k + f(row-1, col+1)) & row > 0 \wedge col = 0 \\ +time[row][col] & \\ min(k + f(row-1, col-1), f(row-1, col)) & row > 0 \wedge col = m-1 \\ +time[row][col] & \end{cases}$$

In the function, the 2D array named time contains the matrix of the weights of each node. An example of the program in action to better explain the algorithm is given below.

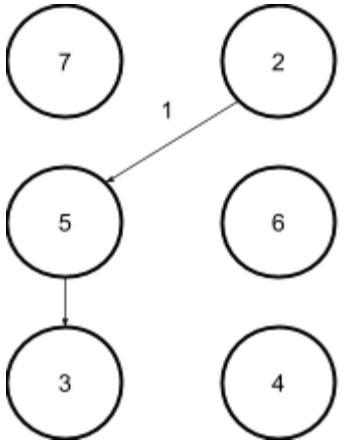


k = 1

Using the left diagram, with a bottom-up approach, we start with the goal back to the base cases. In this example, we start with 4. We have to decisions we can make, either to go to 5 or 6. Though, if we were to move columns to go to node 6, that would incur an additional cost of 1. Therefore we go to node 5. From node 5, we can go to 7 and 2. With the same concept, the final values would be 7 versus 3 -- the 3 is the minimum in this comparison, there we go to the right column to node 2.

The memo table would look like this based on our traversal, which returns the optimal route given to achieve the minimum total distance, found in the right figure.

TRAVERSAL	
	2
8	
11	



k = 1

minimum distance = 11

Problem Kuja-D: How Many Ways (To Edit)?
Status: Accepted
Difficulty Rating: 2
Solution Sketch/Write-Up/Narrative:

The problem will check for the number of ways a given string B can be converted to match with string A. To edit a certain string, the program can either Insert a new character at the current pointer, Delete the character at the current pointer, Copy the current pointed character from String A to String B, or Convert the current pointed character of String B to the current pointed character in String A. A group of these edits is called an edit sequence. The program is then tasked to count the number of possible edit sequences given user input of String A and String B.

To monitor these changes, a memo table is created which dimensions of the length of String A plus 1 by the length of String B plus 1. For example, if the length of String A is 2 and String B is 2, the memo table will be of 3 x 3. The goal is to find the number of ways the pointer can get from (3,3) to (0,0). To represent the possible edits to be made in the table, we can say that inserting moves the pointer to the left node of the current node, deleting moves the pointer to the top node of the current node, and converting or inserting moves the pointer to the top-left node of the current node. In other words, inserting is row - 1, col, deleting is row, col - 1, converting and insertion is row - 1, col - 1. These are the transitions of the program. Now we consider the states of the program. The pointer can be in the left most column, therefore only deleting will be allowed. The pointer can also be in the top most column, therefore only inserting will be allowed because this means that the length of String B is already 0, therefore no deletion and conversion will be possible. Though, in a general case, where the pointer is in the middle section of the matrix, all edits are possible. Like any other Dynamic Programming Bottom-up algorithm, these will be recursively done therefore the base case, which is when row and col reaches the goal which is (0,0), will be equal to 1. The logic behind this is that these 1s will be added to fill up the memo table of the current cell at current row and current col position. These will be done recursively until the goal's cell is added using the sum of the cells at the left, top-left, and top of the goal cell. This gives us a recurrence function of:

$$f(row,\ col) = \begin{cases} 1 & row = 0 \wedge col = 0 \\ f(row-1, col-1) + f(row-1, col) + f(row, col-1) & row > 0 \wedge col > 0 \\ f(row-1, col) & row > 0 \wedge col = 0 \\ f(row, col-1) & row = 0 \wedge col > 0 \end{cases}$$

To illustrate the program, for example, we are given with two strings AB and BA. Our task is to edit String B as many times as possible for it to match string A. We are given the with the memo table of:

		A	B
	1		
B	1		
A	1		Goal

To fill up the first cell, since we are at (0,0) or the base case, we immediately put 1. This is because there is 1 way to edit nothing to nothing, which is 1. Now we, check how many ways we can convert B to nothing, which 1 through deletion (going down). Same goes with A to nothing.

		A	B
	1	1	
B	1	3	
A	1	5	Goal

Now, we can move on to column A. By following the same logic, we then count the number of ways to reach that certain point through going down, right, or diagonal.

		A	B
	1	1	1
B	1	3	5
A	1	5	Goal

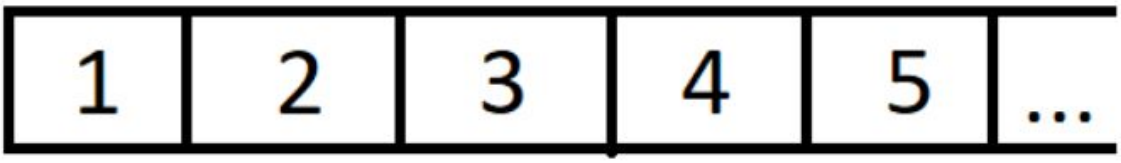
And finally, in column B. We simply just add, the top, top-left, and left node of the current to get it's weight.

Using this completed, memo table we can compute for the Goal's weight, which is 3 + 5 + 5 = 13.

Therefore, there are **13 possible edit sequences.**

Problem Jecht-D: Blasting Through Super Sonic Speed (% Christian)
Status: WRONG
Difficulty Rating: 4
Solution Sketch/Write-Up/Narrative:

The problem involves Sonic going through a straight track consisting of N squares. He starts at square 1 and needs to go to the N'th square as fast as possible or the minimum amount of time and with Sonic having 0 Energy . The inputs consist of (N, A, B, C, K)



- N = is the goal square.
- A= is the time to go to the next square using normal speed.
- B= is the time to go to the next square using spin dash on a non-slippery square.
- C= is the time to get back up after falling due to spin dashing on a slippery square.
- K= is the number of slippery squares

Sonic can either RUN (Go Normally) which will take A seconds and increase his Energy by 1 or DASH (If he has at least 1 Energy) which will take B seconds. The problem will try to look for the optimal approach needed to reach the goal. These are the possible scenarios that Sonic may undergo:

A. If he [RUN] to the next square > Then (n-1 , e+1) meaning He is closer to the goal and he's energy increments by 1 and [Time = Time + A]
B. If he [DASH] to the next Square > (n-1 , e-1) , He will be closer but Energy ill be reduced by 1, [Time = Time + B]
C. If he [DASH] to the next Square but the current Square he is in is {Slippery} > (n, e-1) . In this situation if he dashes he will slip and will not move but will also lose energy because of [DASH] . [Time = Time + C].
D. If Sonic is at the beginning [Time = 0] . (n-1, e=0)

Above is the recurrence function of the program. By looking at the situation I tried using a "COMPLETE SEARCH" approach , we can determine an ideal move per move (between Run VS Dash (if applicable)) and take and store the move that will take less time. But I could not apply the slippery factor for the squares while coding for the But in the accepted code's

approach which uses BitSet and an Array[n][n] does the same thing but it was able to take into account the [SLIP] scenarios.

Example. Input : 5 2 1 4 1 3

6		7			s5
	5				s4
3		4			s3
	2				s2
0					s1

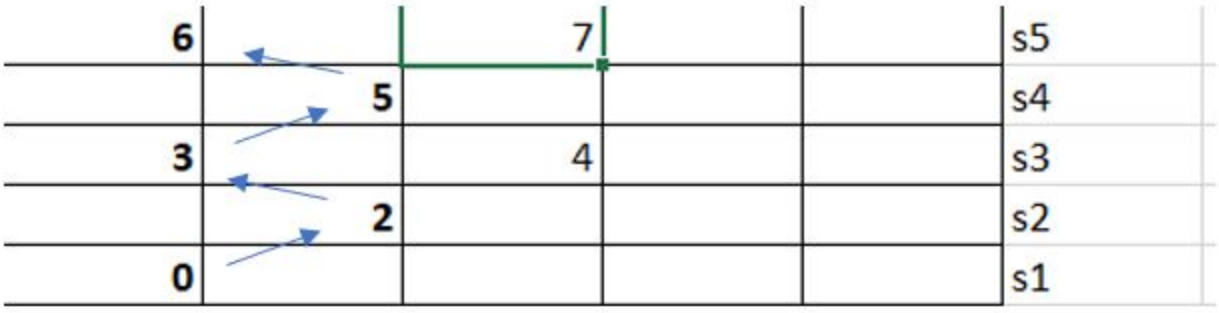
In this image shows the optimal approach for the input. The time needed to get from S1 to S5 here is 6 seconds.

S1 -> S2 ;
Time = 0 ; Energy = 0
RUN > 0 + a (2) = 2 ; <Time = 2 , Energy = 1> *

S2 -> S3 ;
Time = 2 ; Energy = 1
RUN > 2 + a (2) = 4 < Time = 4 , Energy = 2>
DASH > 2 + b(1) = 3 < Time =3 , Energy = 0> *

S3 -> S4 ;
Time = 3 ; Energy = 0
RUN > 3 + a (2) = 5 < Time = 5 , Energy = 1 > *

S4 -> S5 ;
Time = 5 ; Energy = 1
RUN > 5 + a (2) = 7 < Time = 4 , Energy = 2>
DASH > 5 + b(1) = 6 < Time =6 , Energy = 0> *



Problem Jecht-C: All Hail The King (% Sean)
Status: (Choose 1: ACCEPTED/WRONG ANSWER/TIME LIMIT EXCEEDED/RUNTIME ERROR/COMPILE ERROR :(/MEMORY LIMIT EXCEEDED)
Difficulty Rating: (Give this problem a difficulty rating from 1 to 5 with 1 being easiest and 5 being very difficult.)
Solution Sketch/Write-Up/Narrative:

Problem Statement:

We have a King, which in chess, can move in the 8 compass directions. We want to get the King to a certain point on an 8x8 board. The solution we seek to solve in this problem is to count the number of ways we can get to a certain coordinate in this board given the starting position of the king in the board given a certain amount of moves.

Given the problem statement, we can now begin our problem solving process;

First, we decompose the problem into more mathematical terms. We can redefine this problem as given a coordinate, and a set of possible moves (up, up left, up right, right, left, down left, and down right), and a given amount of moves, how many moves does it take to get to the given coordinate. This allows us to realize a few tenets for solving the problem

1. We need to check all the possible routes towards a coordinate given a starting coordinate
2. We need to keep track of how many moves we have left and our current location
3. After every move, the amount of moves left and our current coordinate will change.
4. We can take 8 possible moves

Since this is a counting problem, and in line with the given tenets, we can realize that this is a dynamic programming problem. A dynamic programming problem can be seen if we can apply:

Memoization, which will help solve tenet 1
States, as defined by tenet 2
Transitions, as defined by tenets 3 and 4.

Therefore, our memoization table is more than appropriate since it can directly represent our chessboard. What we save at each coordinate in the memoization table will simply be the number of ways to get to the location given the coordinates and the amount of moves left, which represent our state. Our transitions will be our movement as defined by the set of movements we can take. We can now define the recurrence function.

RECURRENCE

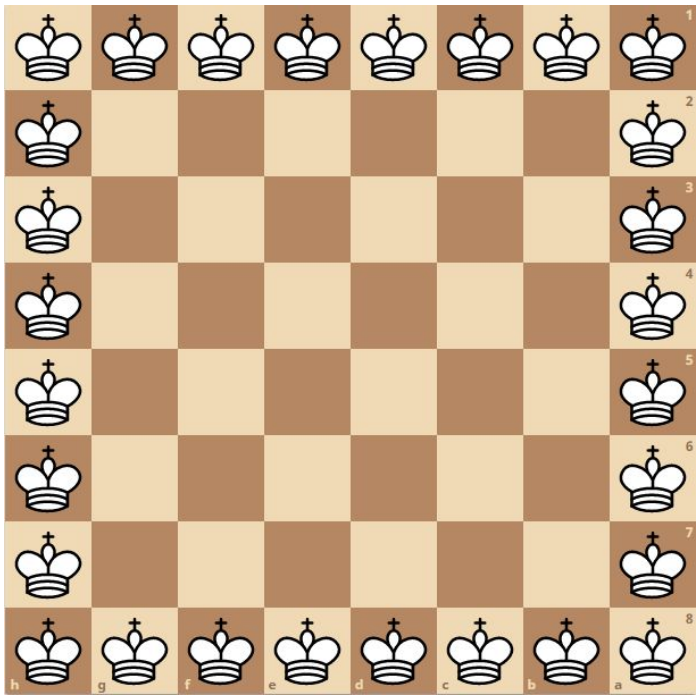
To define our recurrence, we need to take the current coordinates of the King in consideration.

For our base cases, if the king is at the coordinate with 0 moves left, there is only one way for him to reach our target coordinate.

If the king is not at the coordinate with no moves left, there is no way for him to get to the coordinate.

If the king is not at the coordinate but has moves left, we need to check all his possible moves and see how many ways to get him to the destination.

This is the crux of our recurrence, but there is one more state we need to consider:
If the King is in either the corners, the left bound column, the right bound column, the top bound column, or the bottom bound column, the king cannot legally move there. We can see those positions as below:



There are two ways of fixing this problem; we can either set a base case saying that if the king's coordinates are outside the board, there is no way for him to get to the destination, or we can check if the king is in the regions with limited moves and only check the moves that he can take from there. The former is an optimization in terms of runtime, but the later will save a lot of programming headaches.

Given our recurrence cases, we can define our recurrence as such;

$$f(r,c,m)\begin{cases} 1 & r = sr\ and\ c = sr\ and\ m = 0 \\ 0 & r\ != sr\ or\ c\ != sr\ and\ m > 0 \\ 0 & r < 0\ or\ c < 0\ or\ r > 7\ or\ c > 7 \\ \\ f(r-1,c-1,m-1) + f(r+1,c+1,m-1) & \\ +f(r,c+1,m-1) + f(r,c-1,m-1) & \\ +f(r+1,c,m-1) + f(r-1,c,m-1) & \\ +f(r+1,c-1,m-1) + f(r-1,c+1,m-1) & \end{cases}$$

We then define our memoization table as a 3d array, since we need to save the moves left as well.

This solution has a complexity of O(n*m).