PW SKILLS

Lesson Plan

DP-5

PW SKILLS

Today's checklist

- Leetcode 322: Coins
- Bad luck island
- Mpilot problem
- Shortest common supersequence

Leetcode 322: Coins

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.
You may assume that you have an infinite number of each kind of coin.

Example 1:
Input: coins = [1,2,5], amount = 11
Output: 3

Explanation: 11 = 5 + 5 + 1

Example 2:
Input: coins = [2], amount = 3
Output: -1

Example 3:
Input: coins = [1], amount = 0
Output: 0

Constraints:
- 1 <= coins.length <= 12
- 1 <= coins[i] <= 231 - 1
- 0 <= amount <= 104

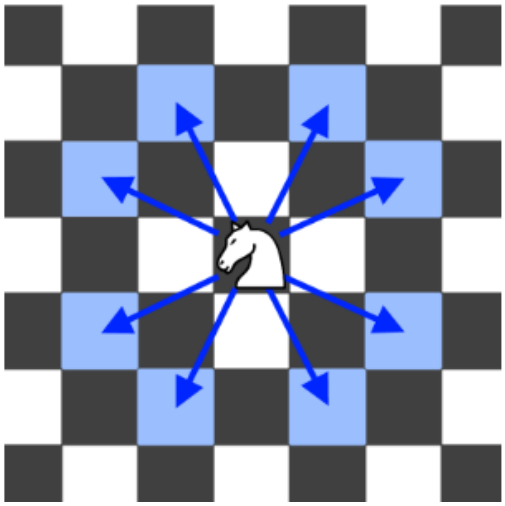
Code:

```
//recursion (top-down approach)
//TC = O(n^amount)
//n = length of coins array

class Solution {
    public int coinChange(int[] coins, int amount) {

        int ans = coinCount(coins, amount);
        return (ans == Integer.MAX_VALUE) ?  -1 : ans;
    }

    int coinCount(int[] coins, int amount) {

        if(amount == 0) {
            return 0;
        }
        if(amount < 0) {
            return Integer.MAX_VALUE;
        }

        int minCoins = Integer.MAX_VALUE;

        for(int i = 0; i < coins.length; i++) {
            int ans = coinCount(coins, amount - coins[i]);

            if(ans != Integer.MAX_VALUE) {
                //we have returned 0 in ans, so now we are updating
the ans count
                //hence 1 + ans
                minCoins = Math.min(minCoins, 1 + ans);
            }
        }
        return minCoins;
    }
}

//recursion with memoization (top-down approach)
//TC = O(n*amount)
//SC = O(amount)
class Solution {

    int[] dp;
    public int coinChange(int[] coins, int amount) {

        dp = new int[amount + 1];
        Arrays.fill(dp, -1);
        int ans = coinCount(coins, amount);
        return (ans == Integer.MAX_VALUE) ?  -1 : ans;
    }
```

```
    int coinCount(int[] coins, int amount) {

        if(amount == 0) {
            return 0;
        }
        if(amount < 0) {
            return Integer.MAX_VALUE;
        }

        if(dp[amount] != -1) {
            return dp[amount];
        }

        int minCoins = Integer.MAX_VALUE;
        for(int i = 0; i < coins.length; i++) {
            int ans = coinCount(coins, amount - coins[i]);

            if(ans != Integer.MAX_VALUE) {

                //we have returned 0 in ans, so now we are updating
the ans count
                //hence 1 + ans
                minCoins = Math.min(minCoins, 1 + ans);
            }
        }
        return dp[amount] = minCoins;
    }
}


//tabulation method (bottom-up approach)
//TC = O(n*amount)
//SC = O(amount)

class Solution {

    public int coinChange(int[] coins, int amount) {

        int[] dp = new int[amount + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for(int i = 1; i <= amount; i++) {
            for(int j = 0; j < coins.length; j++) {

                if(i - coins[j] >= 0 && dp[i - coins[j]] !=
Integer.MAX_VALUE) {
                    dp[i] = Math.min(dp[i], 1 + dp[i - coins[j]]);
                }
            }
        }
        return (dp[amount] == Integer.MAX_VALUE) ?  -1 :
dp[amount];
    }
}

//tabulation method cannot be further space optimised in this case
```

Leetcode 688 Knight Probability in Chessboard

On an n x n chessboard, a knight starts at the cell (row, column) and attempts to make exactly k moves. The rows and columns are 0-indexed, so the top-left cell is (0, 0), and the bottom-right cell is (n - 1, n - 1).
A chess knight has eight possible moves it can make, as illustrated below. Each move is two cells in a cardinal direction, then one cell in an orthogonal direction.



Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.
The knight continues moving until it has made exactly k moves or has moved off the chessboard.
Return the probability that the knight remains on the board after it has stopped moving.

Example 1:

Input: n = 3, k = 2, row = 0, column = 0
Output: 0.06250

Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the board.
From each of those positions, there are also two moves that will keep the knight on the board.
The total probability the knight stays on the board is 0.0625.

Example 2:

Input: n = 1, k = 0, row = 0, column = 0
Output: 1.00000

Constraints:

- 1 <= n <= 25
- 0 <= k <= 100
- 0 <= row, column <= n - 1

Intuition

- The code uses dynamic programming to solve the knight probability problem. The probability array is initialized to all zeros, and then the probability of the knight being on the board after each move is calculated. The base case is that the knight is already on the board, so the probability is 1. The recursive step is to iterate over all possible moves and update the probability of the knight being on the board after each move. The final probability is the probability that the knight is on the board after k moves.

Approach

- The code below uses the DP approach. This approach is more efficient than recursion because it stores the probabilities for moves that have already been made.
  a step-by-step approach for the optimized code to find the probability that the knight remains on the board after making exactly k moves:

1. Initialize a 2D dp array of size n x n to store the probabilities of the knight being on the board from a specific cell.
2. Set the initial probability of the starting cell (row, column) to 1.0, as the knight starts from this cell.
3. For each move from 1 to k, do the following:
   a. Create a new 2D newDp array to store the updated probabilities after one move.
   b. Iterate through each cell (r, c) in the original dp array:
   - For each possible move of the knight, calculate the new position (nr, nc) after the move.
   - If the new position (nr, nc) is within the bounds of the n x n chessboard, update the probability in newDp[r][c] by adding the probability from the original dp[nr][nc] / 8.0. This is because the knight moves uniformly at random, so each move has a probability of 1/8 to be chosen.
   - Set dp to newDp to keep the updated probabilities for the next iteration.
4. After completing all k moves, calculate the total probability of the knight remaining on the board by summing up all the probabilities from the final dp array.
5. Return the total probability as the result.

- The isValid helper function is used to check whether a move is valid, i.e., whether the new position (r, c) after the move is within the bounds of the n x n chessboard. This function returns true if the move is valid, and false otherwise.

Complexity

- **Time complexity:** O(n^2 * k).
  This is because the outer loop iterates over all possible values of k, and the inner loop iterates over all possible rows and columns.
- **Space complexity:** O(n^2)

Code:

```
class Solution {
    private final int[][] moves = {{-2, -1}, {-2, 1}, {-1, -2},
{-1, 2}, {1, -2}, {1, 2}, {2, -1}, {2, 1}};

    public double knightProbability(int n, int k, int row, int
column) {
        double[][] dp = new double[n][n];
        dp[row][column] = 1.0;

        for(int move = 1; move<=k; move++) {
            double[][] ndp = new double[n][n];
            for(int r = 0; r<n; r++) {
                for(int c = 0; c<n; c++) {
                    for(int[] m: moves) {
                        int nr = r+m[0];
                        int nc = c+m[1];
                        if (isValid(nr, nc, n)) ndp[r][c] += dp[nr]
[nc]/8.0;
                    }
                }
            }
            dp = ndp;
        }

        double prob = 0.0;
        for (int r = 0; r < n; r++) {
            for (int c = 0; c < n; c++) {
                prob += dp[r][c];
            }
        }

        return prob;
    }

    private boolean isValid(int r, int c, int n) {
        return r >= 0 && r < n && c >= 0 && c < n;
    }
}
```

Bad Luck Island Codeforces 540D

The Bad Luck Island is inhabited by three kinds of species: r rocks, s scissors and p papers. At some moments of time two random individuals meet (all pairs of individuals can meet equiprobably), and if they belong to different species, then one individual kills the other one: a rock kills scissors, scissors kill paper, and paper kills a rock. Your task is to determine for each species what is the probability that this species will be the only one to inhabit this island after a long enough period of time.

Input
The single line contains three integers r, s and p (1 ≤ r, s, p ≤ 100) — the original number of individuals in the species of rock, scissors and paper, respectively.

Output
Print three space-separated real numbers: the probabilities, at which the rocks, the scissors and the paper will be the only surviving species, respectively. The answer will be considered correct if the relative or absolute error of each number doesn't exceed 10 - 9.

Examples

input
2 2 2
output
0.333333333333 0.333333333333 0.333333333333

input
2 1 2
output
0.150000000000 0.300000000000 0.550000000000

input
1 1 3
output
0.057142857143 0.657142857143 0.285714285714

Approach
Let's count the values dp[r][s][p] — the probability of the situation when r rocks, s scissors and p papers are alive. The initial probability is 1, and in order to calculate the others we should perform the transitions.
Imagine we have r rocks, s scissors and p papers. Let's find the probability of the rock killing scissors (the other probabilities are calculated in the same way). The total number of the possible pairs where one species kills the other one is rs + rp + sp, and the number of possible pairs (rock, scissors) is rs. As all meetings are equiprobable, the probability we want to find is $\frac{rs}{rs+rp+sp}$. This is the probability with which we go the the state dp[r][s − 1][p], with the number of scissors less by one.
In the end, for example, to get the probability of the event that the rocks are alive, we should sum all values dp[i][0][0] for i from 1 to r (the same goes to the other species).

Code:

```
import java.util.Scanner;

public class Main {
    static double[][][] a = new double[105][105][105];

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int R = scanner.nextInt();
        int S = scanner.nextInt();
        int P = scanner.nextInt();

        a[R][S][P] = 1;
        for (int sum = R + S + P; sum > 0; sum--) {
            for (int r = R; r >= 0; r--) {
                for (int s = S; s >= 0; s--) {
                    int p = sum - r - s;
                    if (p < 0 || p > P) continue;
                    if (r == 0 && s == 0) continue;
                    if (r == 0 && p == 0) continue;
                    if (s == 0 && p == 0) continue;
                    double cur = a[r][s][p];
                    int waysR = r * s;
                    int waysS = s * p;
                    int waysP = p * r;
                    int totalWays = waysR + waysS + waysP;
                    if (r > 0) a[r - 1][s][p] += cur * waysP /
totalWays;
                    if (s > 0) a[r][s - 1][p] += cur * waysR /
totalWays;
                    if (p > 0) a[r][s][p - 1] += cur * waysS /
totalWays;
                }
            }
        }

        double ansR = 0;
        double ansS = 0;
        double ansP = 0;
        for (int r = 1; r <= R; r++) ansR += a[r][0][0];
        for (int s = 1; s <= S; s++) ansS += a[0][s][0];
        for (int p = 1; p <= P; p++) ansP += a[0][0][p];

        System.out.printf("%.12f %.12f %.12f\n", ansR, ansS, ansP);
    }
}
```

SPOJ MPILOT

Charlie acquired airline transport company and to stay in business he needs to lower the expenses by any means possible. There are N pilots working for his company (N is even) and N/2 plane crews needs to be made. A plane crew consists of two pilots - a captain and his assistant. A captain must be older than his assistant. Each pilot has a contract granting him two possible salaries - one as a captain and the other as an assistant. A captain's salary is larger than assistant's for the same pilot. However, it is possible that an assistant has larger salary than his captain. Write a program that will compute the minimal amount of money Charlie needs to give for the pilots' salaries if he decides to spend some time to make the optimal (i.e. the cheapest) arrangement of pilots in crews.

Input
The first line of input contains integer N, $2 \leq N \leq 10,000$, N is even, the number of pilots working for the Charlie's company. The next N lines of input contain pilots' salaries. The lines are sorted by pilot's age, the salaries of the youngest pilot are given the first. Each of those N lines contains two integers separated by a space character, X i Y, $1 \leq Y < X \leq 100,000$, a salary as a captain (X) and a salary as an assistant (Y).

Output
The first and only line of output should contain the minimal amount of money Charlie needs to give for the pilots' salaries.

input
4
5000 3000
6000 2000
8000 1000
9000 6000

output
19000

input
6
10000 7000
9000 3000
6000 4000
5000 1000
9000 3000
8000 6000

output
32000

Code:

```
import java.util.Scanner;

public class Main {
    static int[][] dp = new int[2][100005];
    static int[] a = new int[100005];
    static int[] b = new int[100005];
    static int cur = 0;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        for (int i = 0; i < n; i++) {
            a[i] = scanner.nextInt();
            b[i] = scanner.nextInt();
        }

        dp[1][0] = 0;
        for (int i = 1; i <= n / 2; i++) {
            dp[cur][0] = dp[(cur + 1) % 2][0] + b[i - 1];
            for (int j = 1; j <= i; j++) {
                if (j == i)
                    dp[cur][j] = dp[cur][j - 1] + a[i + j - 1];
                else
                    dp[cur][j] = Math.min(dp[cur][j - 1] + a[i + j
- 1], dp[(cur + 1) % 2][j] + b[i + j - 1]);
            }
            cur = (cur + 1) % 2;
        }
        System.out.println(dp[(cur + 1) % 2][n / 2]);
    }
}
```

1092. Shortest Common Supersequence

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there are multiple valid strings, return any of them.
A string s is a subsequence of string t if deleting some number of characters from t (possibly 0) results in the string s.

Example 1:
Input: str1 = "abac", str2 = "cab"
Output: "cabac"

Explanation:
str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".
str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".
The answer provided is the shortest such string that satisfies these properties.

Example 2:

Input: str1 = "aaaaaaaa", str2 = "aaaaaaaa"
Output: "aaaaaaaa"

Constraints:

- 1 <= str1.length, str2.length <= 1000
- str1 and str2 consist of lowercase English letters.

Code:

```
public String shortestCommonSupersequence(String s1, String s2) {
    // find the LCS of s1 and s2
    char lcs[] = lcs(s1,s2).toCharArray();
    int i=0,j=0;
    StringBuilder sb = new StringBuilder();
    // merge s1 and s2 with the LCS
    for(char c:lcs){
        // add characters from s1 until the LCS character is found
        while(s1.charAt(i)≠c) sb.append(s1.charAt(i++));
        // add characters from s2 until the LCS character is found
        while(s2.charAt(j)≠c) sb.append(s2.charAt(j++));
        // add the LCS character
        sb.append(c);
        i++;
        j++;
    }
    // add any remaining characters from s1 and s2
    sb.append(s1.substring(i)).append(s2.substring(j));
    // return the merged string
    return sb.toString();
}

// helper method to find the LCS of two strings
String lcs(String s1,String s2){
    int m = s1.length(),n=s2.length();
    int dp[][] = new int[m+1][n+1];
    // fill the dp array using dynamic programming
    for(int i=1;i≤m;i++){
        for(int j=1;j≤n;j++){
            if(s1.charAt(i-1)==s2.charAt(j-1)){
                dp[i][j]=1+dp[i-1][j-1];
            }else dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);
        }
    }
    // backtrack from the bottom right corner of the dp array to
find the LCS
    StringBuilder sb = new StringBuilder();
    int i=m,j=n;
    while(i>0 && j>0){
        if(s1.charAt(i-1)==s2.charAt(j-1)){
            sb.append(s1.charAt(i-1));
            i--;
            j--;
        }else if(dp[i-1][j]>dp[i][j-1]) i--;
        else j--;
    }
    // reverse the LCS string and return it
    return sb.reverse().toString();
}
```

PW SKILLS

THANK YOU !