Session 14 — Airport Security Task

Note: I do not guarantee the correctness of these solutions! Please double check!

1) wir nutzen Linearität vom Erwartungswert:

$$P[\text{check bag } i] = p_i$$

$$X_i = \begin{cases} 0 & \text{not kept} \quad p_i \cdot r_i \\ 1 & \text{bag } i \text{ is kept} \end{cases}$$

$$P[X_i = 1] = [1-p_i] + p_i \cdot (1-r_i)$$

$$= 1 - p_i \cdot r_i$$

$$E[\bar{X}] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} 1-(p_i \cdot r_i)$$

---

2) Wir nutzen Satz von Bayes und Satz der totalen Wahrscheinlichkeit

$K=1$

$$P[X_1 = 0 \mid X_2 = 0] \qquad 1 \cdot r[1] \qquad p[2] \cdot r[i]$$

$$= \frac{P[X_1 = 0 \cap X_2 = 0]}{P[X_2 = 0]} = \frac{P[X_2 = 0 \mid X_1 = 0) \cdot P[X_1 = 0]}{P[X_2 = 0]}$$

$$P[X_2 = 0 \cap X_1 = 0] + P[X_2 = 0 \cap X_1 = 1]$$
$$P[X_2 = 0 \mid X_1 = 0] \cdot P[X_1 = 0] + P[X_2 = 0 \mid X_1 = 1] \cdot P[X_1 = 1]$$

3) Wir nutzen Rekursion Trick:

$$X = \sum_{i=1}^{n} X_i \qquad X_i = \begin{cases} 1 & \text{wir behalten den i-ten bag} \\ 0 & \text{otherwise} \end{cases}$$

Zu zeigt: Wir suchen $E[X]$. Kann nicht mehr mit Linearität so leicht, da $E[X_i] = P[X_i = 1]$ nicht mehr leicht zu berechnen ist.

Hängt davon ab wie viele Vorgänger bags wir schon haben etc.

Deswegen anderer Approach:

$$E[X] = \mathbb{E}[X \mid X_1 = 0] \cdot Pr[X_1 = 0] + E[X \mid X_1 = 1] \cdot Pr[X_1 = 1]$$

Wieso geht das?

$$E[X] = \sum_{\omega \in \Omega} X(\omega) \cdot P(\omega)$$

$$= \sum_{\omega \in A} X(\omega) \cdot P(\omega) + \sum_{x \in \bar{A}} X(\omega) \cdot P(\omega)$$

$$= \left(\frac{1}{P(A)} \sum_{\omega \in A} X(\omega) \cdot P(\omega) \right) \cdot P(A) + \left(\frac{1}{P(\bar{A})} \cdot \sum_{x \in \bar{A}} X(\omega) \cdot P(\omega) \right) \cdot P(\bar{A})$$

We can rewrite the sum and take the intersection of ω and A in head

$$= \left(\frac{1}{P(A)} \sum_{\omega \in \Omega} X(\omega) \cdot P(\omega \cap A) \right) \cdot P(A) + \left(\frac{1}{P(\bar{A})} \cdot \sum_{x \in \Omega} X(\omega) \cdot P(\omega \cap \bar{A}) \right) \cdot P(\bar{A})$$

$$= \left(\sum_{\omega \in \Omega} X(\omega) \cdot \frac{P(\omega \cap A)}{P(A)} \right) \cdot P(A) + \left(\sum_{x \in \Omega} X(\omega) \cdot \frac{P(\omega \cap \bar{A})}{P(\bar{A})} \right) \cdot P(\bar{A})$$

$$= \left(\sum_{\omega \in \Omega} X(\omega) \cdot \frac{P(\omega \cap A)}{P(A)} \right) \cdot P(A) + \left(\sum_{x \in \Omega} X(\omega) \cdot \frac{P(\omega \cap \bar{A})}{P(\bar{A})} \right) \cdot P(\bar{A})$$

$$= \left(\sum_{\omega \in \Omega} X(\omega) \cdot P(\omega | A) \right) \cdot P(A) + \left(\sum_{x \in \Omega} X(\omega) \cdot P(\omega | \bar{A}) \right) \cdot P(\bar{A})$$

$$= \left(\sum_{\omega \in \Omega} X(\omega) \cdot P(\omega | A) \right) \cdot P(A) + \left(\sum_{x \in \Omega} X(\omega) \cdot P(\omega | \bar{A}) \right) \cdot P(\bar{A})$$

$$= E[X|A] \cdot P(A) + E[X|\bar{A}] \cdot P(\bar{A}) \quad \square$$

Nun haben wir also:

$$E(X) = E(X | X_1 = 0) \cdot P[X_1 = 0] + E[X | X_1 = 1] \cdot P[X_1 = 1]$$

Das ist unsere Rekursion! Erweitern wir das so oft wir:

$$E(X) = E(X | X_1 = 0) \cdot P[X_1 = 0] + E[X | X_1 = 1] \cdot P[X_1 = 1]$$

$$E[X | X_1 = 0, X_2 = 0] \cdot P[X_2 = 0, X_1 = 0] + E[X | X_1 = 0, X_2 = 1] \cdot P[X_2 = 1, X_1 = 0]$$

$$\vdots \qquad \vdots$$

irgendwann sind alle $X_1, \dots, X_n$ festgelegt. → Das ist der Base Case.
Wenn wir genau wissen was passiert ist, können wir die Berechnung leicht machen.

Lösungsansatz:

Wir "simulieren" das Experiment auf alle möglichen Wege, bis wir beim letzten Bug ankommen. Dabei können wir:

$$E[X | X_1 = 0, X_2 = 1] = E[X_1 + X_2 + \sum_{i=3}^{n} X_i \mid X_1 = 0, X_2 = 1] = (0 + 1 + E[\sum_{i=3}^{n} X_i | X_1 = 0, X_2 = 1])$$

vereinfachen. Müssen uns also nur merken, bei welchem Bug wir sind und wie viele Kanalkultur bugs wir schon hatten!

Am Ende müssen wir noch die Rekursion mit Memoization verbessern, weil wir oft den selben Wert berechnen und ihn einfach speichern können!

Lösung: solve(0,0)

```
// curBag: at which bag we are now in the "simulation"
// consecutive: how many consecutive bags we've taken
solve(int curBag, int consecutive) {
    if (curBag == n) return 0;   // End event

    // case where we check all bags
    if (consecutive ≥ K) {
        return (0 + solve(curBag+1, consecutive)) · Pr[Take bag 'curBag' | K consecutive bags]   // keep it
             + (1 + solve(curBag+1, 0)) · Pr[not take bag 'curBag' | K consecutive bags];       // reset it
             // = 1 − Pr[take bag 'curBag']
    }

    // case where we don't check all bags
    else {
        return (0 + solve(curBag+1, consecutive+1)) · Pr[check bag 'cur bag'] ·     // they take it; not K consecutive bags
               Pr[take bag 'cur Bag']
             + (1 + solve(curBag+1, 0)) · (1 − Pr[check bag 'cur Bag'] · Pr[take bag 'cur Bag'])   // they don't take it; reset
    }
}
```

Check if this solves the sample case. Yes, it does! :)

Now, we only need to change the code a little bit to use memoization and make it fast!

Changed code with memoization table (2 dimensional because our method has two arguments)

```
double[][] memo = new double[n][K+1]   // n: amount of bags
                                       // K+1: highest value for consecutive bags. After K bags taken we just keep it at K
solve(int curBag, int consecutive) {
    if (curBag == n) return 0;   // End event

    // new case: we already computed the value and want to reuse it
    if (memo[curBag][consecutive] ≠ -1) return memo[curBag][consecutive];

    // case where we check all bags
    if (consecutive ≥ K) {
return  memo[curBag][consecutive] = (0 + solve(curBag+1, consecutive)) · Pr[Take bag 'curBag' | K consecutive bags]   // keep it
             + (1 + solve(curBag+1, 0)) · Pr[not take bag 'curBag' | K consecutive bags];   // reset it
    }

    // case where we don't check all bags
    else {
return memo[curBag][consecutive] = (0 + solve(curBag+1, consecutive+1)) · Pr[check bag 'cur bag'] ·   // they take it; not K consecutive bags
               Pr[take bag 'cur Bag']
             + (1 + solve(curBag+1, 0)) · (1 − Pr[check bag 'cur Bag'] · Pr[take bag 'cur Bag'])   // they don't take it; reset
    }
}
```

Note that we initialize the memo array with -1 to distinguish whether we already computed the value or not!
Note that we can declare the memo array and any other data we use as static to use it within the solve method.

Wir müssen uns kurz Kopf machen, wo die Lösung ist!
Es ist einfach der Wert, der von solve(0,0) returned wird.

Dieser Approach funktioniert mit jeder ABW DP Aufgabe! Probiert es!

Complete Code for Airport Security Task:

```java
static double[][] Memo;
static double[] p;
static double[] r;
static int n;
static int k;

public static void testCase() {
      // Input using In.java class
      int q = In.readInt();
      n = In.readInt();
      k = In.readInt();
      p = new double[n];
      r = new double[n];
      for(int i=0;i<n;i++){
        p[i] = In.readDouble();
        r[i] = In.readDouble();
      }
      if (q == 1){
        double sum = 0;
        for(int i=0;i<n;i++){
          sum += 1 - (p[i]*r[i]);
        }
        Out.println(sum);
      }
      else if (q==2){
        double out = r[1] * p[0]*r[0];
        out /= (out + p[1]*r[1]*(1-p[0]*r[0]));
        Out.println(out);
      }
      else{
        Memo = new double[n][k+1];
        for(int i=0;i<n;i++){
          for(int j=0;j<k+1;j++){
            Memo[i][j] = -1;
          }
        }
        Out.println(solve(0,0));
      }

      // Output using Out.java class

  }

  public static double solve(int curBag, int consecutive){
    if(curBag == n) return 0;
    if(Memo[curBag][consecutive] != -1) return Memo[curBag][consecutive];
    if(consecutive >= k){
      return Memo[curBag][consecutive] = (0 + solve(curBag+1, consecutive) * r[curBag])
          +  (1 + solve(curBag+1, 0)) * (1-r[curBag]);
    }
    else{
      return Memo[curBag][consecutive] = (0 + solve(curBag+1, consecutive+1) *
p[curBag] * r[curBag])
          +  (1 + solve(curBag+1, 0)) * (1-(p[curBag] * r[curBag]));
    }
  }
}
```