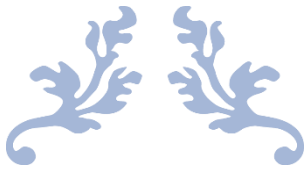

# PROBLEMS: UNIT 4

Dynamic Programming

Javier García Jiménez and Isabel Martínez Gómez

# PROBLEM 3

In problem 3 it is asked to program an algorithm with dynamic programming that returns the minimum banknotes to use to pay some quantity of money.

We have a group of banknotes and we have quantities of each banknote.

With dynamic programming our goal is to make a matrix to calculate the solution.

In the matrix, each position represents the minimum banknotes possible to pay a quantity of money.

For example, if we have banknotes of 1, 2, 5 and 10, we have 4 lines in the matrix.

Each line represents the minimum banknotes possible to pay some money using the banknote of the line and the previous ones. For example, if we are in line 2, we are using banknotes of 5, 2 and 1.

So, in order to calculate the result for each quantity, we must create a matrix of N columns (N is the money) and K lines (each line represents a banknote) we have to consider some things:

- The first column of the matrix, that represents the minimum banknotes to pay 0 euros, is set to 0, because we need 0 banknotes to pay 0 euros.
- The first line of the matrix is set from 1 to N to an infinite amount, because we need infinite banknotes of 0 to pay more than 0 euros.

Once we are on a “normal” position, we have to set the value of the position. First, we set the minimum quantity to the quantity of the same money with the previous banknote. This is because we haven’t calculated yet how many banknotes, we may use to pay the money with the actual banknote.

If j (the column) is less than the value of the banknote that we are looking for, then the value of that position is the value of the previous line (the previous banknote). For example, if we have the banknotes that we had in the previous example, if we are looking to pay 4 euros, and we have already calculated the value for the 1 and 2 banknotes, now we have to calculate the value for the banknote of 5.

Just because 5 is greater than 4, we can’t pay 4 with a banknote of 5, so the minimum banknotes to pay 4 with banknotes of 1,2 and 5 is equal to the minimum banknotes to pay.

The last case is divided in to two subcases. In this case we have a quantity of money in which we need to calculate the minimum number of banknotes taking in account the actual banknote that we are looking at.

The two subcases consist of if there are enough banknotes to pay the money or not.

As we told before, we don’t have unlimited quantity of each banknote.

So, it’s necessary to check if we can pay the quantity with the number of banknotes of the type that we are looking at or not.

We have to take care of this because if the quantity of money that we have to pay is greater than the banknote multiplied by the number of banknotes that we have of that type, It means that we need to add all the banknotes if we have to pay  some money with that type of banknote.

Let's see an example to clarify this.

If we have to pay 6 euros, and we only have 3 banknotes of 1 and 2 of 2, if we are looking at the 2 banknote, 2 * 2 is 4, so using all 2's banknotes is not enough to pay the quantity, so we must add all the 2's banknotes to calculate the value that we are looking for.

This is done in the function called enoughBanknotes.

So, once we have decided this, we must calculate the value of the position in our matrix.

- If we have enough banknotes to pay the quantity, the minimum banknotes necessary to pay the money is:
    - Min (minimumValue (set before), matrix[i][j] – value of the banknote] +1)

Let's see an example of this: if we have to pay 3 euros and we have banknotes of 1 and 2, if we are looking the 2 banknotes, the minimum value to pay 3 euros is:

Min (matrix [i-1][j], matrix[i][j-2]+1) → min ( matrix[1][3], matrix[2][1] + 1) → min(3,1+1) → min(3,2) → **2**

- If we don't have enough banknotes to pay the quantity the minimum banknotes necessary to pay the money is:
    - Min (minimumValue (set before), matrix[i-1][ j – value of the banknote * quantity of the banknote] + quantity of the banknote)

Let's see an example of this: if we have to pay 6 euros and we have 6 banknotes of 1 and 2 banknotes of 2, and we are looking the 2 banknotes, the minimum banknotes to pay is:

Min(matrix[i-1][j], matrix[i-1][j- (2*2)] + 2) → min(matrix[1][6],matrix[1][6-4]+2) →

Min(6,matrix[1][2]+2) → Min(6,4) → **4**

With all of this, we build a matrix with n (number of different banknotes) rows and N (the money that we have to pay). So the solution for our problem is the position (n,N) of out matrix, which represents the minimum number of banknotes possible to pay some money.

Here are some images of the code of this part of the code:

```
public int algorithm(int N) {
    int[][] myTable = new int[banknotes.length][N + 1];

    for (int k = 0; k < banknotes.length; k++) {
        myTable[k][0] = 0;
    }
    for (int k = 1; k <= N; k++) {
        myTable[0][k] = 9999;
    }

    for (int i = 1; i < banknotes.length; i++) {
        for (int j = 1; j <= N; j++) {
            int min = myTable[i - 1][j];
            if (i == 1 && j < banknotes[i]) {
                myTable[i][j] = 9999;
            } else if (j < banknotes[i]) {
                myTable[i][j] = myTable[i - 1][j];
            } else {
                if (enoughBanknotes(j, i)) {
                    myTable[i][j] = Math.min(min, myTable[i][j - banknotes[i]] + 1);
                } else {
                    myTable[i][j] = Math.min(min, myTable[i-1][j - banknotes[i] * quantities[i]] + quantities[i]);
                }
            }
        }
    }
```

The problem asked also to return which banknotes we have used to compose the payment, so our function hadn't end yet.

In order to get what banknotes, we have used, we have to go through the matrix that we have created, decomposing the composition of banknotes that he did in the first part of the function.

First, we have created some auxiliary variables that allows us to have some control on the banknotes that we are going through. Here they are:

```
int price = N;
int[] quantAux = new int[quantities.length];
for(int j = 0; j< quantities.length;j++)
{
    quantAux[j] = quantities[j];
}
int i = banknotes.length-1;
ArrayList<Integer> used = new ArrayList<>();
```

We have a variable price, that stores the money, because we need to add banknotes until we have no money left.

Then we have an array of the quantities of the banknotes. This is made in order to get how many banknotes have we used of each type. In the loop we copy the values in quantities into quantAux, which is the auxiliary array.

Last but not least, we have created and array that stores the type of banknotes that we have used.

Let's start with how we decompose the composition of banknotes.

In order to get what banknotes, we have used, we have to go through the matrix until we have no money left to pay, so it's necessary to make a while loop to do this.

Now, we have some scenarios in this case, to add or not add a banknote into out array of banknotes used. Let's see what we have here:

- First, if the price is lesser than the value of the banknote, we can't add that banknote, so we go to the next. For example, if we have to pay 8 euros and we are looking in the banknote of 10, we can't add that banknote to the array.
- If not, the banknote must be added to our array of used banknotes:
    - If the banknote is not added yet, we add it into out array
    - If the money minus the value of the banknote is still greater than the value of the banknote, but we only have 1 banknote of that type left, it means that we won't have enough banknotes of that type in the next iteration. So, we need to extract one more quantity of that banknote, and we have to go through the next banknote because we got out of stock of the actual banknote and extract the value of the banknote to the price.
    - In any other case, we have to extract one quantities of the banknote and we have to extract the value of the banknote to the price.

This is the code:

```
while(price != 0 && i>0)
{
    if(price-banknotes[i] <0)
    {
        i--;
    }
    else
    {
        if(!used.contains(banknotes[i]))
        {
            used.add(banknotes[i]);
        }
        if((price-banknotes[i]) >= banknotes[i] && quantAux[i] == 1)
        {

            quantAux[i]=quantAux[i]-1;
            price = price-banknotes[i];
            i--;
        } else
        {
            quantAux[i]=quantAux[i]-1;
            price = price-banknotes[i];
        }
    }
}
```

Lastly, the next few lines of code, are dedicated to print in screen, the banknotes and the quantities of the banknotes:

```
for(int k = 0; k< used.size();k++)
{
    int banknote = used.get(k);
    int quant ;
    if(banknote == 0) {
        quant = 0;
    }else if(banknote == 1)
    {
        quant = 1;
    }
    else if(banknote == 2)
    {
        quant = 2;
    }
    else if(banknote == 5)
    {
        quant = 3;
    }else
    {
        quant = 4;
    }
    System.out.println("Banknote of: " + banknote +" --> ("+(quantities[quant]-quantAux[quant])+")");
}
```

With this, we get what number of the banknote we have used, this is because in the array of banknotes, 1 is 1, 2 is 2, 3 is 5 and 4 is 10. So, we need this to identify inside of the matrix, what banknote we have used. This is also because we need to extract the quantities aux that we have made, to the original quantities, so we can get what quantities of each banknote we have used.

For 25 euros, having these banknotes:

```
private final int[] banknotes = {0,1,2,5,10};
private final int[] quantities = {0,10,5,2,1};
```

This is the result:

```
Banknote of: 10 --> (1)
Banknote of: 5 --> (2)
Banknote of: 2 --> (2)
Banknote of: 1 --> (1)
```

With all of this, we have a function programmed with dynamic programming, that returns the minimum banknotes possible to pay some money and what banknotes must be used (specifying quantities) to pay that money.

# PROBLEM 6

In this exercise we have to calculate how many money will win Javi Potter win with a bet in a tournament of 4 teams.

To do this we have to use dynamic programming, so our goal is to build a matrix of 4 rows (the number of teams) and N columns (the number of games they have to win).

Javi Potter will bet for The Griffins, which in our algorithm will be the first team of the matrix.

In order to build the matrix, we need to consider some things.

- The probability to win money if Griffins have 0 wins left (they won the tournament) is 1. If any other team has 0 wins left it means that Griffins have not win, so the probability to win money is 0.

This means that in our 4 dimension matrix, when we are on position (0,j,k,h) it means that Griffins have won, so the value of that is always 1. Otherwise, if any other team has 0 games left to win, that position is 0. Here is the code:

```
for(int i = 0; i< games;i++)
{
    for(int j = 0; j < games; j++)
    {
        for(int k = 0; k< games ; k++)
        {
            for(int h = 0 ; h < games; h++)
            {
                myTable[0][j][k][h] = 1;
                myTable[i][0][k][h] = 0;
                myTable[i][j][0][h] = 0;
                myTable[i][j][k][0] = 0;
            }
        }
    }
}
```

So we have set all the probabilities when one teams have won the tournament, now we have to calculate the probabilities in intermediate stages of the tournament, so when we arrive to the final, we know what is the probability for Griffins to win the tournament.

These probabilities are based on a value called VC, which is the Quality Value of the teams. With this, we know that if a team have much more VC than the others, it's more likely for that team to win a game.

So, taking in count all of this, we can calculate the probability of a team to win N games.

This is known by making a simple statistic/probability operation.

We have 4 teams, so we have 4 different probabilities which form 1 in total.

So, the probability in any stage of the game where anyone has won the tournament is calculated with this:

```
myTable[i][j][k][h] = probA * myTable[i-1][j][k][h] + probB * myTable[i][j-1][k][h]
                      +probC * myTable[i][j][k-1][h] +probD * myTable[i][j][k][h-1];
```

probA, probB, probC and probD are known at the start of the program thanks to the VC values explained before.

With this, if we look the position of the matrix [N-1] [N-1] [N-1] [N-1] we will look at the possibilities that the Griffins have to win N games.

So, if we have the money that Javi Potter will bet and the probabilities for the Griffins to win, if we divide money/probability, we will know how much money will Javi win:

```
return money/myTable[games-1][games-1][games-1][games-1];
```

If we have this data:

```
Problem6 p6 = new Problem6( p1: 50, p2: 20, p3: 10, p4: 20, nGames: 4, m: 100);
```

P1, p2, p3 and p4 are the probabilities, nGames is the number of games and m is the money bet. We have this solution:

```
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Will win: 135.64104 euros.
```

As we can see, Javi will not win so much money because the probability for the Griffins is much higher than the others. Let's see what happen if we reduce the probability of the Griffins:

```
Problem6 p6 = new Problem6( p1: 10, p2: 60, p3: 10, p4: 20, nGames: 4, m: 100);
```

Let's see the solution:

```
Will win: 5242.496 euros.
```

As we can see, this time Javi wins much more money, because the Griffins have much less probability to win the tourney.