ALGORITMIEK: some solutions to exercise class 9

Problem 2.

We use the algorithm from the lecture slides to fill the knapsack table row-by-row. This yields the following table:

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	65

The maximal value of a feasible subset is $F[5][6] = 65$. The optimal subset is $\{\text{item } 3, \text{item } 5\}$.

Problem 3.

a. As said, $P(i, j)$ is the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series. If team A wins the next game, which happens with probability p, A will need $i - 1$ more wins to win the series while B will still need j wins. If team A looses the game, which happens with probability $q = 1 - p$, A will still need i wins while B will need $j - 1$ wins to win the series. This leads to the recurrence relation:

$$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1) \text{ for } i, j > 0$$

The initial conditions follow immediately from the definition of $P(i, j)$:

$$P(0, j) = 1 \text{ for } j > 0, \; P(i, 0) = 0 \text{ for } i > 0$$

b. Here is the dynamic programming table in question, with its entries rounded-off to two decimal places. (It can be filled either row-by-row, or column-by-column, or diagonal-by-diagonal.)

$i \backslash j$	0	1	2	3	4
0		1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.06	0.18	0.32	0.46
4	0	0.03	0.09	0.18	0.29

c.

```
Algorithm WorldSeries (int n, double p)
// Computes the odds of winning a series of n games
// Input: A number of wins n needed to win the series
//        and probability p of one particular team winning a game
// Output: The probability of this team winning the series
{ q = 1-p;
  for (j=1; j<=n; j++)
```

```
      P[0][j] = 1.0;
    for (i=1; i<=n; i++) {
      P[i][0] = 0.0;
      for (j=1; j<=n; j++)
        P[i][j] = p * P[i-1][j] + q * P[i][j-1];
    }

    return P[n][n];
  }
```

Problem 4.
a. For the recurrent formulation, we make the following observation: As hinted in the problem statement, the weight g_n may or may not be in the subset we search for. If g_n is in the subset, then we must form the remaining weight $W - g_n$ from the remaining weights $g_1, \ldots, g_{n-1}$. If g_n is not in the subset, then we must form the full weight W from the remaining weights. Hence, W can be formed from $g_1, \ldots, g_n$, if and only if $W - g_n$ can be formed from $g_1, \ldots, g_{n-1}$, or W can be formed from $g_1, \ldots, g_{n-1}$.
The case that $n = 1$, i.e., that we have only one weight g_1, is the base case of the recursion. In that case, the answer to the problem is yes, i.e., W can be formed from weight g_1, if and only if $W = 0$ or $W = g_1$.
Let the array `weight` be global, and let $g_1, \ldots, g_n$ be stored as `weight[1],...,weight[n]`. We then have the following recursive function:

```
  bool balance (int n, int W)
  { if (n==1)
    { if (W==0 || W==weight[1])
        return true;
      else
        return false;
    }
    else  // n >= 2
    { bool withThisWeight = balance (n-1, W-weight[n]);
      bool withoutThisWeight = balance (n-1, W);
      if (withThisWeight || withoutThisWeight)
        return true;
      else
        return false;
    }
  }
```

It depends on the actual weights g_i, whether there will be little or much overlap between the subproblems.
Note that the second argument of the recursive call `balance (n-1, W-weight[n])` may be negative. This does not hurt. In that case, the value returned will always be false.
b. The array element `weighing[i][j]` should be true, if and only if the subproblem with weights $g_1, \ldots, g_i$ and total weight j has a solution. The answer to the original problem is then represented by `weighing[n][W]`. Note that for $i = 1$, only `weighing[1][0]` and `weighing[1][g_1]` are true. We then have the following recurrence relation (where `w` is

short for `weighing`):

$$\texttt{w[i][j]} = \begin{cases} \texttt{w[i-1][j-weight[i]]} \text{ or } \texttt{w[i-1][j]} & \text{if } i > 1 \text{ and } j \geq \texttt{weight[i]} \\ \texttt{w[i-1][j]} & \text{if } i > 1 \text{ and } j < \texttt{weight[i]} \\ \text{true} & \text{if } i = 1 \text{ and } (j = 0 \text{ or } j = \texttt{weight[1]}) \\ \text{false} & \text{if } i = 1 \text{ and } (j \neq 0 \text{ and } j \neq \texttt{weight[1]}) \end{cases}$$

Note that, as a result, `weighing[i][0]` is true for $i > 1$. Note also that, unlike with part **a.**, we must prevent a negative index $j -$ `weight[i]`. Therefore, we consider the case that $j \geq$ `weight[i]` and the case that $j <$ `weight[i]`, separately.
c. The value of `weighing[i][j]` can be obtained from (one or) two elements of the previous row, namely `w[i-1][j-weight[i]]` and `w[i-1][j]`. Hence, we can fill the array row by row, from top to bottom. It does not matter whether we fill a row from left to right or from right to left.
It is also possible, though less natural, to fill the array column by column. In that case, we must fill the columns from left to right, and each column must be filled from top to bottom.
In the algorithm below, we choose to fill the array row by row, and each row from left to right.

```
void balance2 (int n, int W)
{ int i, j;
    // fill row 1:
  for (j=1;j<=W;j++)
    weighing[1][j] = false;
  weighing[1][0] = true;
  weighing[1][weight[1]] = true;
    // fill rows 2,...,n:
  for (i=2; i<=n; i++)
  { for (j=0;j<=W; j++)
    { if (j>=weight[i])
        weighing[i][j] = weighing[i-1][j-weight[i]] || weighing[i-1][j];
      else
        weighing[i][j] = weighing[i-1][j];
    }
  }
}  // balance2
```

Note that the entries in row 0 of array `weighing` remain unused.
Both the time complexity and the space complexity of the algorithm are in $\Theta(n * W)$, because we compute $n * W$ array elements and each element is computed in constant time.
Note that this dynamic programming approach would not work, if the weights were non-integer. The recursive solution from part **a.** would still work for non-integer weights.
d. Start in `weighing[n][W]`. If that value is true (and only then), there is a solution. In that case, we build up the subset of weights, as follows:
If `weighing[n-1][W]` is also true, then g_n does not have to be included in the solution. Otherwise, we must have $W \geq$ `weight[n]`, `weighing[n-1][W-weight[n]]` must be true, and g_n must be included in the solution. In the former case, continue in the same way from `weighing[n-1][W]`. In the latter case, continue from `weighing[n-1][W-weight[n]]`. Hence, if you 'come from above', then do not include g_i. If you 'come from the left', then do include g_i.

Continue in this manner until row $i = 1$. If, by then, $j = 0$, then g_1 must not be included. If $j > 0$, then j must be equal to g_1 and g_1 must be included.
Note that, if at some point during this procedure, both `weighing[i-1][j]` and `weighing[i-1][j-weight[i]]` happen to be true, then there exists a solution without g_i and a solution with g_i. We can freely choose to exclude g_i or to include g_i.
e. If we do not wish to reconstruct the subset, we do not have to store all intermediate results in a 2D array. A 1D array `weighing` of size $W+1$, corresponding to a row in the 2D array, suffices then. In order to compute the value of `weighing[j]` corresponding to row i, we need the 'old value' of `weighing[j-weight[i]]` (provided that $j \geq$ `weight[i]`). We can ensure that this old value with a smaller index is still available, by computing the new values of `weighing[j]` from right to left.

Problem 5.
a. We assume that the rows of the board are numbered 1 to m from bottom to top, and that the columns are numbered 1 to n.

```
double yield (int i, int j)
{ if (i==0)  // fallen out of the board, hence zero yield
    return 0.0;
  else
  { if (board[i][j]=='.')  // ball continues in same column
      return yield (i-1, j);
    else
    { if (board[i][j]=='*')  // ball continues either left or right
        return (0.5 * yield (i-1, j-1) + 0.5 * yield (i-1, j+1));
      else  // a gate
        return board[i][j];
    }
  }
}  // yield

double max = -1.0;
int maxcol = 0;
for (int j=1; j<=n; j++)
{ double thisYield = yield (m, j);
  if (thisYield > max)
  { max = thisYield;
    maxcol = j;
  }
}
```

Note that in this algorithm, the types of the entries of the board are not really consistent: sometimes, we assume they are characters, sometimes we assume they are numbers. In a consistent representation, we may, e.g., encode the characters `'.'` and `'*'` as numbers.
c. Let $D[i][j]$ denote the expected yield, if we drop a ball in row i and column j. Then $D[i][j]$ satisfies the following recurrence relation:

$$D[i][j] = \begin{cases} 0 & \text{if } i = 0 \\ D[i-1][j] & \text{if } i > 0 \text{ and } \texttt{bord[i][j]='.'} \\ 0.5 * D[i-1][j-1] + 0.5 * D[i-1][j+1] & \text{if } i > 0 \text{ and } \texttt{bord[i][j]='*'} \\ \texttt{bord[i][j]} & \text{if } i > 0 \text{ and } \texttt{bord[i][j]} \text{ is een getal} \end{cases}$$

To compute the value of $D[i][j]$ we may need $D[i-1][j]$ or $D[i-1][j-1]$ and $D[i-1][j+1]$. Hence, we must compute the values in row $i-1$ before the values in row i. We therefore compute the entries of array D row by row, from row 0 to row n. It does not matter in which order we compute the entries in a row. The following algorithm computes them from left to right:

```
double yield2 (int m, int n)
{ int i, j;
    // fill row 0:
  for (j=1; j<=n; j++)
    D[0][j] = 0.0;
    // fill rows 1 to m:
  for (i=1; i<=m; i++)
  { for (j=1; j<=n; j++)
    { if (board[i][j]=='.')  // ball continues in same column
        D[i][j] = D[i-1][j];
      else
      { if (board[i][j]=='*')  // ball continues either left or right
          D[i][j] = 0.5 * D[i-1][j-1] + 0.5 * D[i-1][j+1];
        else  // a gate
          D[i][j] = board[i][j];
      }
    }
  }

    // determine maximum value in row m
  double max = -1.0;
  int maxcol = 0;
  for (int j=1; j<=n; j++)
  { if (D[m][j] > max)
    { max = D[m][j];
      maxcol = j;
    }
  }
  return max;

}  // yield2
```

Problem 6. The quantity $C(n,k)$ satisfies the following recurrence relation:

$$C(n,k) = \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0, n \end{cases}$$

a. We can fill a two-dimensional array `C`, where `C[i][j]` $= \binom{i}{j}$, row-by-row with the following bottom-up DP algorithm:

```
int bin(int n,int k) {
    for ( i = 0; i <= n; i++ )
        for ( j = 0; j <= min(i,k); j++ )
```

```
            if ( ( j == 0 ) || ( j == i ) )
                C[i][j] = 1;
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
    return C[n][k];
}
```

We use algorithm `bin` to fill the table. We only fill columns 0–3, because we do not need higher columns to compute $C(6, 3)$. This yields the following numbers:

$i\backslash j$	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
6	1	6	15	20

b. Yes, the table can also be filled column-by-column, with each column filled top-to-bottom starting with 1 on the main diagonal of the table. This is achieved with the following code:

```
int bin4 (int n, int k) {
  for (j=0; j<=k; j++)
    for (i=j; i<=n; i++)
      if (j==0 || j==i)
        C[i][j] = 1;
      else
        C[i][j] = C[i-1][j-1] + C[i-1][j];

  return C[n][k];
}
```