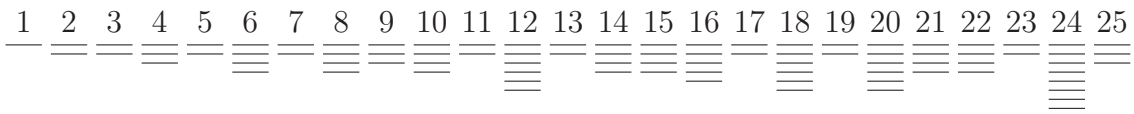
ALGORITMIEK: enkele uitwerkingen bij werkcollege 1

2. After the first iteration, all doors are open. After the second iteration, only the odd-numbered doors are open (the even-numbered doors have just been closed). In the third iteration, the multiples of 3 are toggled, etc.

   We consider the result for an example, in particular the case $n = 25$. Every time we toggle a door, we add an underline:



   Note that the number of times door m is toggled (i.e., the number of lines under door m) equals the number of divisors of m. When this number is odd, the door is open. When this number is even, the door is closed.

   We observe that doors 1, 4, 9, 16 and 25 are open, exactly the squares. This will be the case for every n: the square-numbered doors $\leq n$ will be open.

   This can be explained as follows: for every number m, both 1 and m divide m, so door m is toggled (a.o.) in iterations 1 and m (two times, unless $m = 1$). Whenever k divides m, also $\frac{m}{k}$ divides m, so divisors of m come in pairs. Therefore, in general a number m has an even number of divisors, which means that door m will is closed at the end.

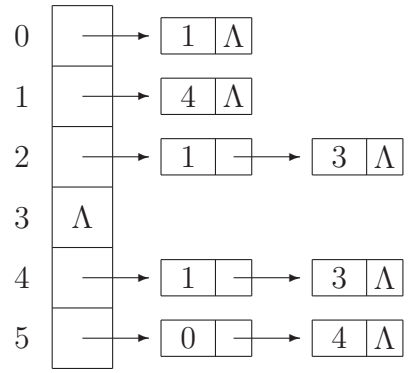
   The only exception are the squares $m = k^2$. For such m, k equals $\frac{m}{k}$, so this particular divisor comes alone. Hence, the total number of divisors of a square m is odd, which means that door m is open at the end.

6. There are four cases to be considered:

   Directed graph, adjacency matrix. In this case, `graaf[i][j]` $= 1$, if and only if there is a *directed* edge from i to j. The matrix `graaf` is not necessarily symmetric. The adjacency matrix for example graph 2 is:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Directed graph, adjacency list. In this case, `graaf[i]` must be interpreted as the header of a list of nodes j for which there is a *directed* edge from i to j. The adjacency list for example graph 2 is:



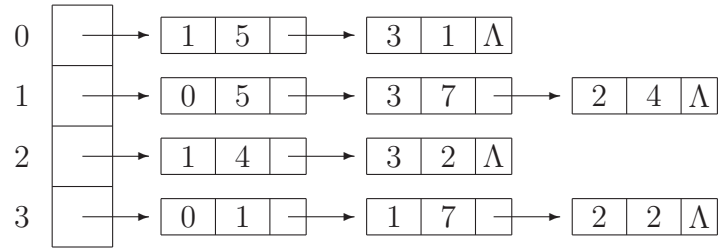
Weighted graph, adjacency matrix. In this case, `graaf[i][j]` is the weight of the edge between i and j (or from i to j). Hence, the matrix `graaf` can hold values different from 0 and 1. We should choose a proper default value for edges that are missing in the graph. Depending on the situation, this might for instance be: 0, ∞, -1 or $-\infty$. When we use ∞ as the default value, the adjacency matrix for example graph 3 is:

$$\begin{pmatrix} \infty & 5 & \infty & 1 \\ 5 & \infty & 4 & 7 \\ \infty & 4 & \infty & 2 \\ 1 & 7 & 2 & \infty \end{pmatrix}$$

Weighted graph, adjacency list. In this case, `graaf[i]` is the header of a list of objects of an extended class `buur`, which also contains the weight of the edge represented:

```
class buur
{ public:
    int knoopnummer;
    int gewicht;
    buur* volgende;
};  // buur
```

The adjacency list for example graph 3 is:



7.
```
void telvoorEuler ( buur* graaf[n] ) {
  int totaal = 0;
  for ( int i=0; i<n; i++ ) {
    teller = 0;
    buur* hulp = graaf[i];
    while ( hulp != NULL ) {  // buurlijst aflopen
      teller ++;
      hulp = hulp->volgende;
    }
    if ( teller%2 == 1 )  // oneven
      totaal ++;
  }  // for
  if ( totaal <= 2 )
    cout << "hooguit twee..." << endl;
  else
    cout << "meer dan twee..." << endl;
}
```

8. (a)
```
int takken ( buur* graaf[n] ) {
  int count = 0;
  for ( int i=0; i<n; i++ ) {
    buur* hulp = graaf[i];
    while ( hulp != NULL ) {  // buurlijst aflopen
      count++;
      hulp = hulp->volgende;
    }
  }
    // alles 'per ongeluk' dubbel geteld -> corrigeren
  return count/2;
}
```

(b)
```
int takken2 ( int graaf[n][n] ) {
  int count = 0;
  for ( int i=0; i<n; i++ ) {
    for ( int j=0; j<n; j++ ) {
      if ( graaf[i][j] > 0 ) {
        count++;
      }
    }
  }
    // alles 'per ongeluk' dubbel geteld -> corrigeren
  return count/2;
}
```

Variant: alleen deel rechtsboven de diagonaal van de adjacency matrix aflopen, of juist alleen het deel linksonder de diagonaal. Dan hoef je na afloop niet door 2 te delen.

(c) De adjacency list representatie is het meest efficiënt, omdat je daarbij alleen maar kijkt naar de bestaande takken. Bij de adjacency matrix representatie loop je ook niet-bestaande takken af.

9. (a) Omkeren van een pijl (i, j) met adjacency matrix `int graaf[n][n]` .
Er loopt tevoren een tak (i, j), maar geen tak (j, i), dus `graaf[i][j] = 1` en `graaf[j][i] = 0`. Dit draaien we om.

```
void draaiom ( int graaf[n][n], int i, int j ) {
  graaf[i][j] = 0;
  graaf[j][i] = 1;
}
```

(b) Omkeren van een pijl (i, j) met adjacency list `buur* graaf[n]` .
De lijsten zijn niet per se gesorteerd. Er loopt tevoren een tak (i, j), maar geen tak (j, i), dus j komt buur in lijst `graaf[i]` en i komt niet voor in lijst `graaf[j]` .

Idee voor algoritme:
- j in buurlijst van i opzoeken
- j uit buurlijst van i verwijderen
- i vooraan in buurlijst van j plaatsen.

```
void draaiom2 ( buur* graaf[n], int i, int j ) {
  buur* hulp = graaf[i];
  buur* vorige = NULL;
  while ( hulp->knoopnummer != j ) { // knoop j zoeken
    vorige = hulp;
    hulp = hulp->volgende;
  }
    // haal buur j uit lijst
  if ( vorige == NULL ) { // j is eerste buur van i
    graaf[i] = hulp->volgende;
  }
  else { // j is niet eerste buur van i
    vorige->volgende = hulp->volgende;
  }
  delete hulp;  // gooi buur j weg

    // voeg nu i vooraan in de buurlijst van j toe
  hulp = new buur;
  hulp->knoopnummer = i;
  hulp->volgende = graaf[j];
  graaf[j] = hulp;
}
```