Programming Fundamentals

- Lecturer: Dr. Arnold Meijster
- e-mail: a.meijster@rug.nl
- room: 5161.343 (Bernoulliborg)

More on `printf`: int format specifiers

`%c`	character
`%d`	signed decimal
`%ld`	long decimal
`%u`	unsigned decimal

```
printf("%c %d %u\n", 'e', -17, 17);
```

Output:
```
e -17 17
```

Formatted Output: printf

- **Examples:**

```
printf("%d%d", 123, 456);  // prints: 123456

printf("%d%4d", 123, 456); // prints: 123 456

printf("%04d", 123);       // prints: 0123

printf("pi=%f and e=%f.\n", 3.1415926, 2.718);

printf("%.1f\n", 1.49);    // prints 1.5
```

More on `printf`: float specifiers

Standard precision is 11 positions (including decimal dot):

```
%f    ddd.ddd
%e    d.dddde{sign}dd
%E    d.ddddE{sign}dd
```

```
printf("%f\n", 1234.56789);
printf("%5.3f\n", 1234.56789);
printf("%12.4e\n", 1234.56789);
```

```
1234.567890
1234.568
  1.2346e+03
```

(at least 5 positions, 3 digits after .)

(at least 12 positions, 4 digits after .)

Coercion: implicit type conversion

Coercion/Widening:

$$\begin{tikzcd}
\text{byte} \arrow[d] & \\
\text{short} \arrow[d] & \\
\text{int} \arrow[r] \arrow[d] & \text{float} \arrow[d] \\
\text{long} \arrow[r] & \text{double}
\end{tikzcd}$$

```
short x;
int y, a;
long b;
float t;
double r;

x = 1234;
y = 5674;
a = x + y;
b = a*x;
t = b/2.25;
r = t + 7;
```

Explicit type conversion: type cast

Coercion/Widening:

$$\begin{tikzcd}
\text{byte} \arrow[d] & \\
\text{short} \arrow[d] & \\
\text{int} \arrow[d] \arrow[r] & \text{float} \arrow[d] \\
\text{long} \arrow[r] & \text{double}
\end{tikzcd}$$

```
short x;
int y, a;
long b;
float t;
double r;

x = 1234;
y = 5674;
x = (short)y - 4*x;
b = y*x;
y = (int)b%y;
t = (float)(b/y);
r = (double)b/y;
y = (int)t;
```

Type conversion: type cast

- Beware! The typecast operator has a very high priority!
  - `(int)3.5*2` yields `6`
  - `(int)(3.5*2)` yields `7`

Example: sum of squares

Exercise: write a program fragment that sums the squares of all natural numbers less than 20.

0*0 + 1*1 + 2*2 + .. + 19*19

```
int sumSquares =
    0*0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 +
    6*6 + 7*7 + 8*8 + 9*9 + 10*10 + 11*11 +
    12*12 + 13*13 + 14*14 + 15*15 + 16*16 +
    17*17 + 18*18 + 19*19;
printf("sumSquares = %d\n", sumSquares);
```

Example: sum of squares

```
int sumSquares = 0;
sumSquares +=   1*1;
sumSquares +=   2*2;
sumSquares +=   3*3;
sumSquares +=   4*4;
sumSquares +=   5*5;
sumSquares +=   6*6;
sumSquares +=   7*7;
sumSquares +=   8*8;
sumSquares +=   9*9;
sumSquares += 10*10;
sumSquares += 11*11;
sumSquares += 12*12;
sumSquares += 13*13;
sumSquares += 14*14;
sumSquares += 15*15;
sumSquares += 16*16;
sumSquares += 17*17;
sumSquares += 18*18;
sumSquares += 19*19;
printf("sumSquares = %d\n", sumSquares);
```

Example: sum of squares

For those who like math, it is well-known that

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

```
int n = 19;
printf("sumSquares = %d\n", n*(n+1)*(2*n+1)/6);
```

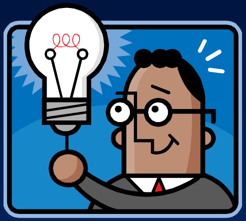
Example: sum of squares

Exercise: write a program fragment that sums the squares of all natural numbers less than 2000

(do not use the formula on the previous slide. It will overflow!)

0*0 + 1*1 + 2*2 + .. + 1999*1999

Example: sum of squares



Clearly, we need some sort of iteration.

C has several loop-constructs. One of them is the for-loop.

Idea: We use a variable `i` to iterate over the range [0..2000) and add `i*i` to `sumSquares`.

```
  int i, sumSquares = 0;
  for (i=0; i < 2000; i++) {
    sumSquares += i*i;
  }
  printf("sumSquares = %d\n", sumSquares);
```

Declaration of the loop variable

```
int i, sumSquares = 0;
for (i=0; i < 2000; i++) {
   sumSquares += i*i;
}
printf("sumSquares = %d\n", sumSquares);
```

```
int sumSquares = 0;
for (int i=0; i < 2000; i++) {
   sumSquares += i*i;
}
// At this point the loop variable i does not exist
printf("sumSquares = %d\n", sumSquares);
```

For-statement

The general syntax of the for-statement is:

```
for (initialisation; condition; update) {
    body;
}
```

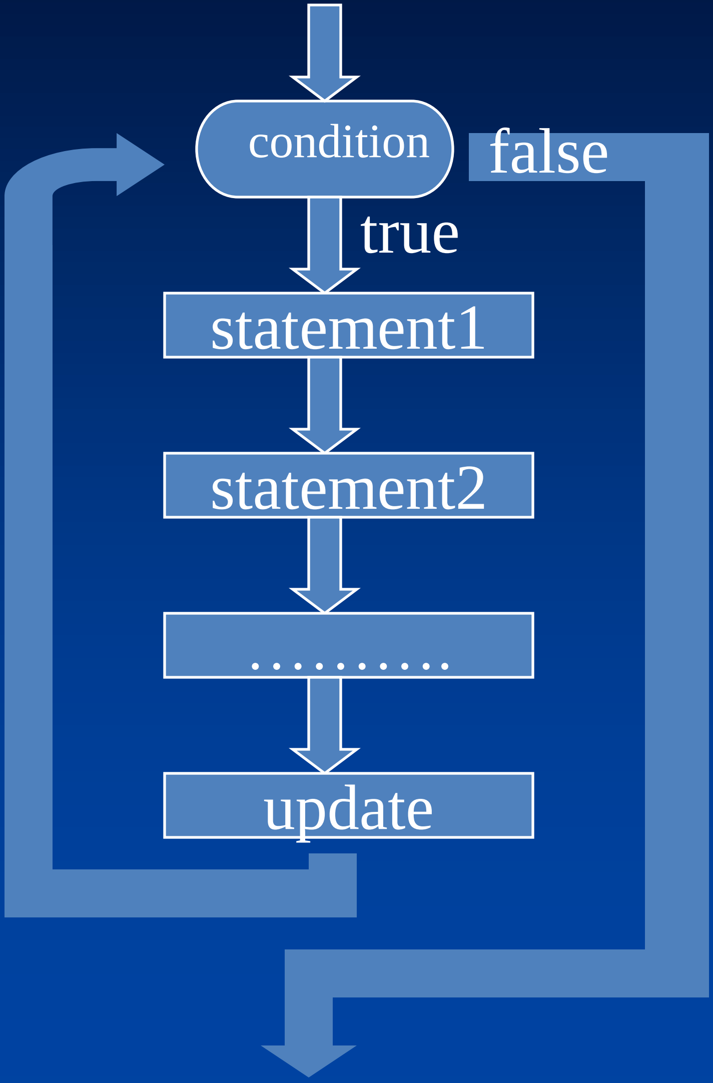
First, the statement `initialisation` is executed.

Before each iteration (also the first one!), the boolean `condition` is evaluated. This condition is also known as the *guard* of the loop. If it is true (i.e. non-zero value), then the statements of the `body` are executed, followed by execution of `update`.

If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the for-loop.

For-statement

```
for (initialisation; condition; update) {
    statement1;
    statement2;
    ..
}
```



Example: it’s full of stars

Exercise: write a program fragment that reads a non-negative integer `n` from the input and prints a series of `n` stars (asterisks, *) on the output.

```
int n;
scanf("%d", &n);
for (int i=0; i < n; i++) {
  putchar('*');      // or printf("*");
}
```

Note that, at the beginning of each iteration, the value of `i` is equal to number of printed stars (so far).

Example: it's full of stars (2)

An alternative solution:

```
int n;
scanf("%d", &n);
for (int i=n; i > 0; i--) {
  putchar('*');
}
```

This time, at the beginning of each iteration, the value of `i` is equal to the number of stars that still need to be printed.

Example: it’s full of stars (3)

A 3rd alternative:

```
int n;
scanf("%d", &n);
for (; n > 0; n--) {
  putchar('*');
}
```

Note that the initialization of the for-loop may be empty.

Example: product of odd integers

Exercise: write a program fragment that computes the product of all odd natural numbers less than some value **`lim`**.

```
int oddProduct = 1; /* note the 1 (not 0) */
for (int i=0; i < lim; i++) {
  if (i%2 == 1) {
    oddProd *= i;
  }
}
```

Example: product of odd integers

A nicer (and more efficient) solution is:

```
int oddProd = 1;
for (int i=3; i < lim; i+=2) {
  oddProd *= i;
}
```

continue-statement

Sometimes we want to ‘early’ continue to the next iteration. The continue statement does exactly that. It can be used with any type of loop: for, while, do-while.

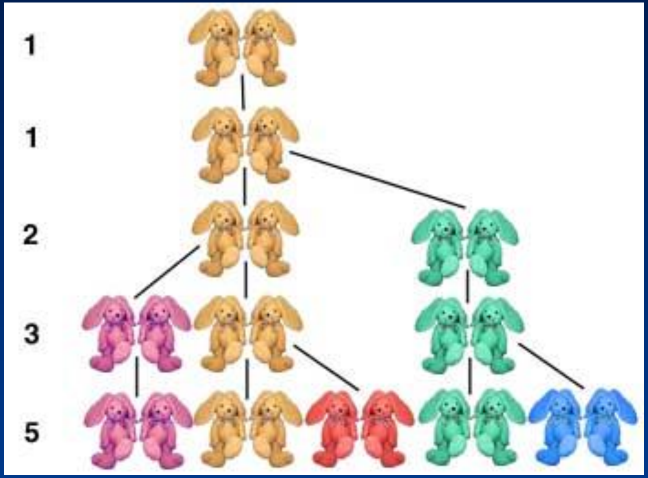
Exercise: write a program fragment that computes the product of all odd natural numbers less than some value `lim` that are not divisible by 7.

```
int oddProd = 1;
for (int i=3; i < lim; i+=2) {
   if (i%7 == 0) {
      continue;
   }
   oddProd *= i;
}
```

Fibonacci: one, one, two, three, five, …

We start with one pair of young rabbits. A rabbit is mature after one year. Each pair of mature rabbits 'produces' one pair of young rabbits:

year	young	mature	total
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8

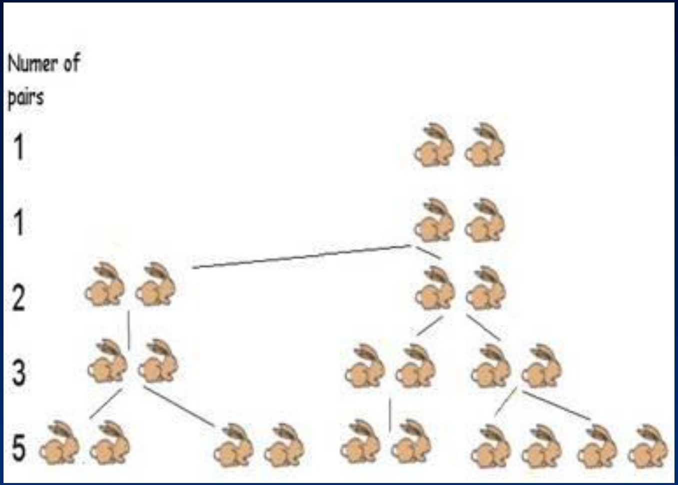


Exercise: write a program fragment that, given a non-negative integer `n`, prints the above table for the years `0` upto `n`:

Note that:

year	young	mature
y	a	b
y + 1	b	a + b

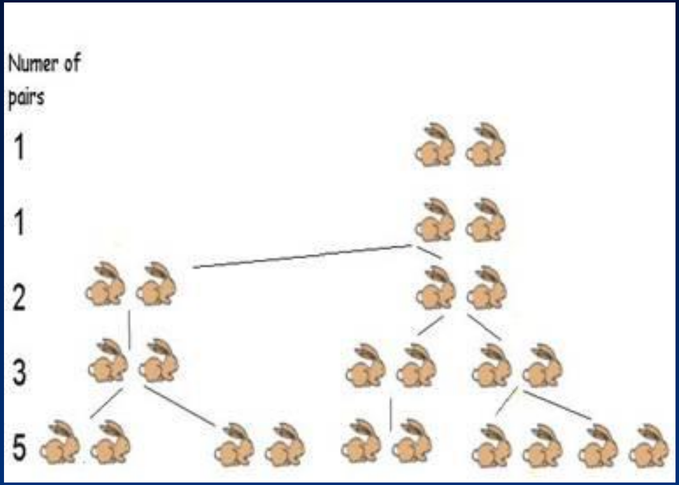
Fibonacci: one, one, two, three, five, …



```
int young = 1;
int mature = 0;

for (int year=0; year < n; year++) {
  printf("%d\t%d\t%d\t%d\n",
         year, young, mature, young + mature);
    /* young == A, mature == B */
    /* young == A, young + mature == A + B */
  mature = young + mature;
    /* young == A, mature == A + B */
    /* mature - young == B, mature == A + B */
  young = mature - young;
    /* young == B, mature == A + B */
}
```

Fibonacci: one, one, two, three, five, …



```
int young = 1;
int mature = 0;

for (int year=0; year < n; year++) {
  printf("%d\t%d\t%d\t%d\n",
         year, young, mature, young + mature);
  mature = young + mature;
  young = mature - young;
}
```

Fibonacci: one, one, two, three, five, …

Variation: In which year do we reach a total number of pairs that is at least (a given) `n`?

```
int young = 1;
int mature = 0;
int year = 0;
int n;
scanf("%d", &n);
while (young + mature < n) {
  mature = mature + young;
  young = mature - young;
  year++;
}
/* here: young + mature >= n  (logical negation of guard) */
printf("In year %d we have at least %d pairs.\n", year, n);
```

While-statement

The syntax of the while-statement is:

```
while (condition) {
    body;
}
```

Before each iteration (also the first one!), the boolean `condition` is evaluated. If it is true (i.e. non-zero), then the statements of the `body` are executed.

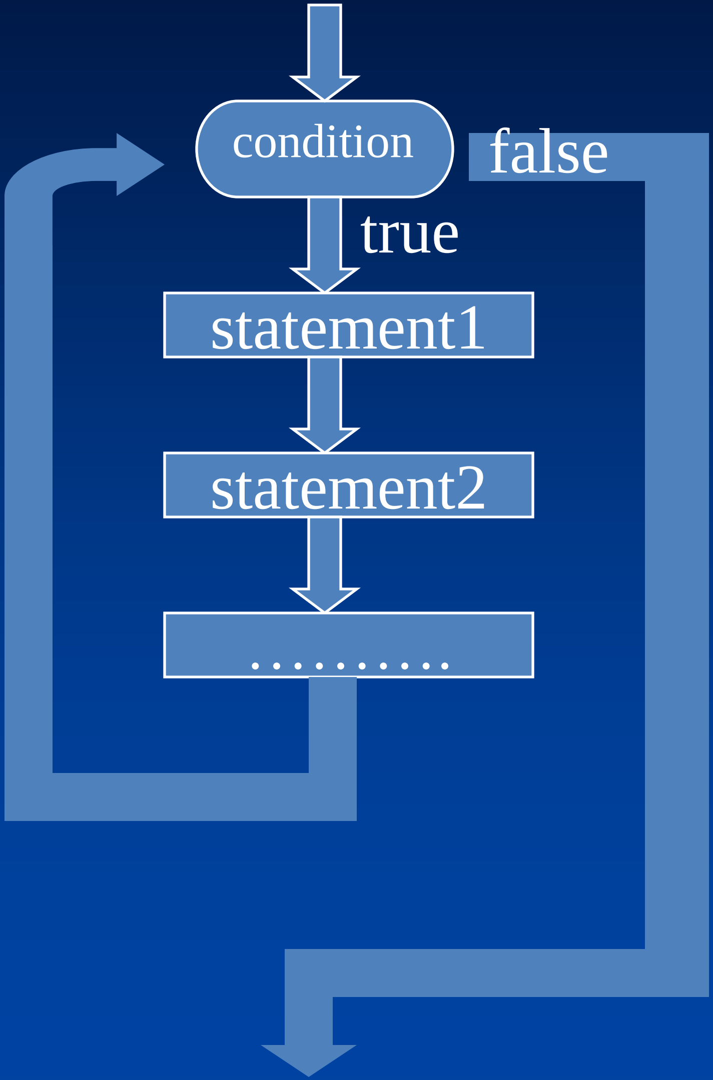
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the while-loop.

The while-loop and the for-loop are very similar. In fact, the general form above can be written as a for-loop with an empty initialisation and an empty update (not very stylish):

```
for (; condition; ) {
    body;
}
```

While-statement

```
while (condition) {
     statement1;
     statement2;
     ..
   }
```



Summing the input

Exercise: write a program fragment that reads a series of integers from the input, and outputs the sum of the numbers. The series is terminated by a zero.

```
int number, sum = 0;

scanf("%d", &number);
while (number != 0) {
  sum += number;
  scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

```
int number, sum = 0;

scanf("%d", &number);
while (number) {
  sum += number;
  scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

Do-While-statement

The syntax of the do-while-statement is:

```
do {
   body;
} while (condition);
```

First, the `body` is executed (without testing the `condition`).
Note that the body of the loop is executed at least once!

At the end of each iteration, the boolean `condition` is evaluated. If it is true (i.e. non-zero), then the statements of the `body` are executed again.

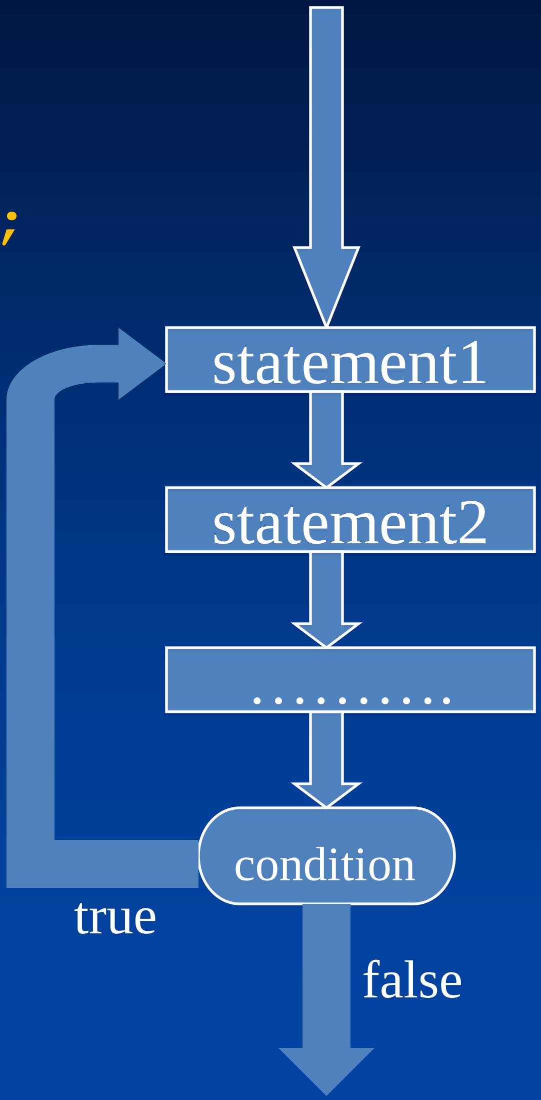
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the do-while-loop.

Equivalent with:

```
body;
while(condition){
  body;
}
```

Do-While-statement

```
do {
      statement1;
      statement2;
      ..
} while (condition) ;
```



Summing the input (II)

Exercise: write a program fragment that reads a series of integers from the input, and outputs the sum of the numbers. The series is terminated by a zero.

```
int number, sum = 0;

scanf("%d", &number);
while (number) {
  sum += number;
  scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

```
int number, sum = 0;

do {
  scanf("%d", &number);
  sum += number;
} while (number);
printf("sum=%d\n", sum);
```

Example: read input

Ask the user to type in a positive integer ≥ 1:

```
int number;

do {
  printf ("Please, type an integer >=1: ");
  scanf("%d", &number);
  if (number < 1) {
    printf("Number is not >= 1, try again.\n");
  }
} while (number < 1);
```

`break`-statement

Sometimes we want to ‘break’ out of a loop.

In C you can do this with the `break`-statement.
It can be used with any type of loop: for, while, do-while.

It is almost always used in combination with an if-statement.

A `break`-statement has the same effect as normal loop termination: the loop stops, and execution of the program continues directly after the loop.

Example: grade calculator

For some course, a teacher has a list of integer grades. There are three grades per student. To help him calculate the final grades, we write a program (fragment) that repeatedly reads three integer valued grades `(x, y, z)` from the input and then prints the weighted average using the factors 0.2, 0.3 and 0.5. The program should stop if the entered value for `x` is zero.

```
int x;
float grade;
while (1) {  /* infinite loop, alternative is for(;;) */
  printf("1st grade: ");
  scanf("%d", &x);
  if (x == 0) {  /* alternative test: !x */
    break;
  }
  grade = 0.2*x;
  printf("2nd grade: ");
  scanf("%d", &x);
  grade += 0.3*x;
  printf("3rd grade: ");
  scanf("%d", &x);
  grade += 0.5*x;
  printf ("Weighted average: %2.1f\n", grade);
}
```

Which loop to use?

This is a matter of style! Each of the three loop-constructs can be expressed as the other two. So, you could say that they are equivalent.

Still, there is a preference based on style arguments:

for-statement: use it when we know the number of iterations in advance.

while-statement: use it when we do not know the number of iterations in advance.

do-while-statement: use it when we do not know the number of iterations in advance and the body of the loop must be performed at least once.

Collatz: 3n+1 problem

The Collatz conjecture is an unsolved conjecture in mathematics named after Lothar Collatz, who first proposed it in 1937 at Syracuse University. The conjecture is also known as the $3n + 1$ problem, or the Syracuse problem.

Collatz conjecture

The Collatz conjecture says that the following algorithm will eventually terminate for any non-negative integer *a*.

```
1. Print $a$
2. If $a = 1$ goto step 4
3. If $a$ is even then
      divide $a$ by two,
   otherwise
      set $a \leftarrow 3a + 1$
   Go to Step 1.
4. Print “Done”
```

Collatz conjecture

```
1. Print $a$
2. If $a = 1$ goto step 4
3. If $a$ is even then
      divide $a$ by two,
   otherwise
      set $a \leftarrow 3a + 1$
   Print $a$
     Go to Step 1.
4. Print "Done"
```

```
int main(int argc, char *argv[]) {
   int a;
  scanf("%d", &a);
  printf("%d", a);
  while (a != 1) {
     a = (a%2 ? 3*a + 1 : a/2);
     printf(", %d", a);
  }
  printf("\nDone\n");
  return 0;
}
```

Collatz conjecture

Mathematicians aren’t so interested in the actual sequences, but rather the number of terms in the sequence until you get to 1

For example,

```
27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445,
1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425,
1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858,
2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40,
20, 10, 5, 16, 8, 4, 2, 1     [112 iterations]
```

```
171, 514, 257, 772, 386, 193, 580, 290, 145, 436, 218, 109, 328,
164, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445,
1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425,
1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858,
2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40,
20, 10, 5, 16, 8, 4, 2, 1    [125 iterations]
```

Collatz conjecture: number of steps

```
int main(int argc, char *argv[]) {
  int a;
  scanf("%d", &a);
  int steps = 0;
  while (a != 1) {
    a = (a%2 ? 3*a + 1 : a/2);
    steps++;
  }
  printf("%d steps\n", steps);
  return 0;
}
```

Prime number?

Write a program fragment that reads a non-negative integer n from the keyboard and outputs YES if n is a prime number, and NO otherwise.

Prime number? A naive solution

```
int n;
scanf("%d", &n);
int a = 2;
while (a < n) {
   if (n%a == 0) {
      break; // n is not a prime
   }
   a++;
}
printf((n >= 2) && (a == n) ? "YES\n" : "NO\n");
```

The maximum 32 bit signed integer is $2^{31} - 1 = 2147483647$, and happens to be a prime. This means that for this number, the above program takes 2,147,483,647 iterations!

Prime number?

Problem analysis:

1. if there exists integers $a > 1$ and $b > 1$ such that $a \cdot b = n$ then n is not a prime number, otherwise it is.
2. We may assume that $a \leq b$, otherwise we swap the roles of a and b.
3. This means that $a^2 \leq a \cdot b$.
4. Hence, if we search for a suitable a using a while-loop, then we can stop searching once $\mathrm{a}^2 > n$.
5. The logical negation of $a^2 > n$ is $a^2 \leq n$, so that serves as the test of the loop.

Prime number? A better solution

```
int n;
scanf("%d", &n);
int a = 2;
while (a*a <= n) {
   if (n%a == 0) {
      break; // n is not a prime
   }
   a++;
}
printf((n >= 2) && (a*a > n) ? "YES\n" : "NO\n");
```

Since $\sqrt{2^{31} - 1} \approx 46341$, it will now require 46341 steps to determine that $2^{31} - 1 = 2147483647$ is prime. In other words, this program is expected to be more than 40000 times faster for this particular input!

Prime number? Minor improvements

Note that 2 is the only even prime. So, instead of `a++` we could start with `a=3` and use `a+=2`. This halves the number of iterations.

Moreover, in each iteration we compute `a*a` in the condition `a*a <= n`. On most CPUs multiplication takes more time (cpu clock times) than addition. Moreover, multiplication times 2^n is a very cheap operation (it only requires shifting the bit representation of the number by n positions to the left and extending it with zeros).

Now, realize that: $(a+2)^2 = a^2 + 4 \cdot a + 4$. So, if we maintain in an extra variable `sq` the value of `a*a`, then we can replace the condition by sq<=n, and update `sq` using `sq += 4*a + 4` followed by `a += 2;`

Prime number? Final version

```
int n;
scanf("%d", &n);
int a = 3, sq = 9;
while (sq <= n) {
  if (n%a == 0) {
    break; // n is not a prime
  }
  sq += 4*a + 4;
  a += 2;
}
printf((n==2) || ((n >= 2) && (n%2) && (sq > n))
       ? "YES\n" : "NO\n");
```

Exponentiation: c $= a^b$

```
/* compute a^b, call this value M */
/* a^b==M */
int c = 1;
/* c*a^b==M */
while (b != 0) {
  /* c*a^b==M, b>0 */
  /* c*a*a^(b-1)==M */
  c = c*a;
  /* c*a^(b-1)==M */
  b = b - 1;
  /* c*a^b==M */
}
/* b==0, c*a^b==M */
/* c==M */
```

Exponentiation: $c = a^b$

```
int c = 1;
while (b != 0) {
   c = c*a;
   b = b - 1;
}
```

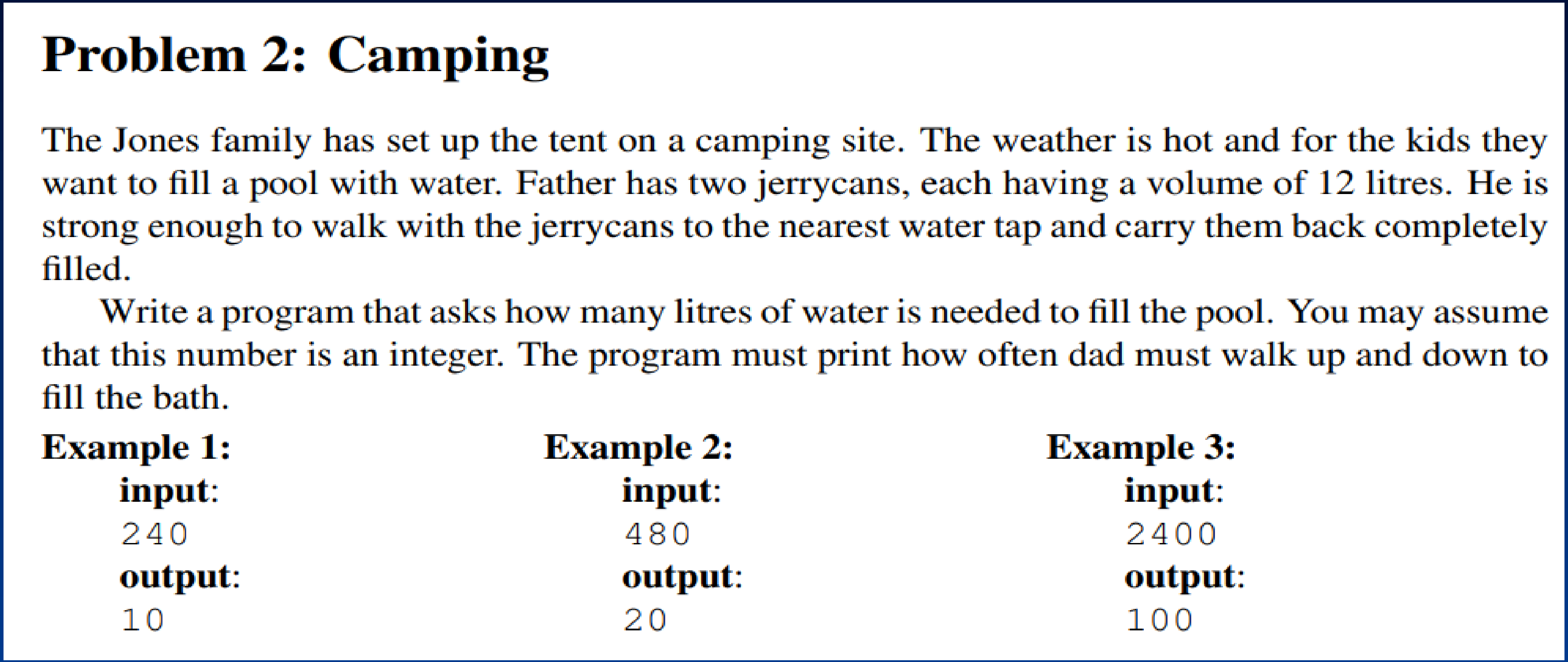
Exponentiation: c $= a^b$ (faster version)

```
int c = 1;
/* c*a^b==M */
while (b != 0) {
  /* c*a^b==M, b>0 */
  if (b%2 == 0) {
    /* c*a^b==M, b>0, b even */
    /* c*(a^2)^(b/2)==M */
    a = a*a;
    /* c*a^(b/2)==M */
    b = b/2;
    /* c*a^b==M */
  } else { // do what we did before
    /* c*a*a^(b-1)==M */
    c = c*a;
    b = b - 1;
    /* c*a^b==M */
  }
  /* c*a^b==M */
}
/* c==M */
```

Exponentiation: $c = a^b$ (faster version)

```
int c = 1;
/* c*a^b==M */
while (b != 0) {
  if (b%2 == 0) {
    a = a*a;
    b = b/2;
  } else {
    c = c*a;
    b = b - 1;
  }
}
/* c==M */
```

Solutions 1st lab session



Problem 2: Camping

The Jones family has set up the tent on a camping site. The weather is hot and for the kids they want to fill a pool with water. Father has two jerrycans, each having a volume of 12 litres. He is strong enough to walk with the jerrycans to the nearest water tap and carry them back completely filled.

Write a program that asks how many litres of water is needed to fill the pool. You may assume that this number is an integer. The program must print how often dad must walk up and down to fill the bath.

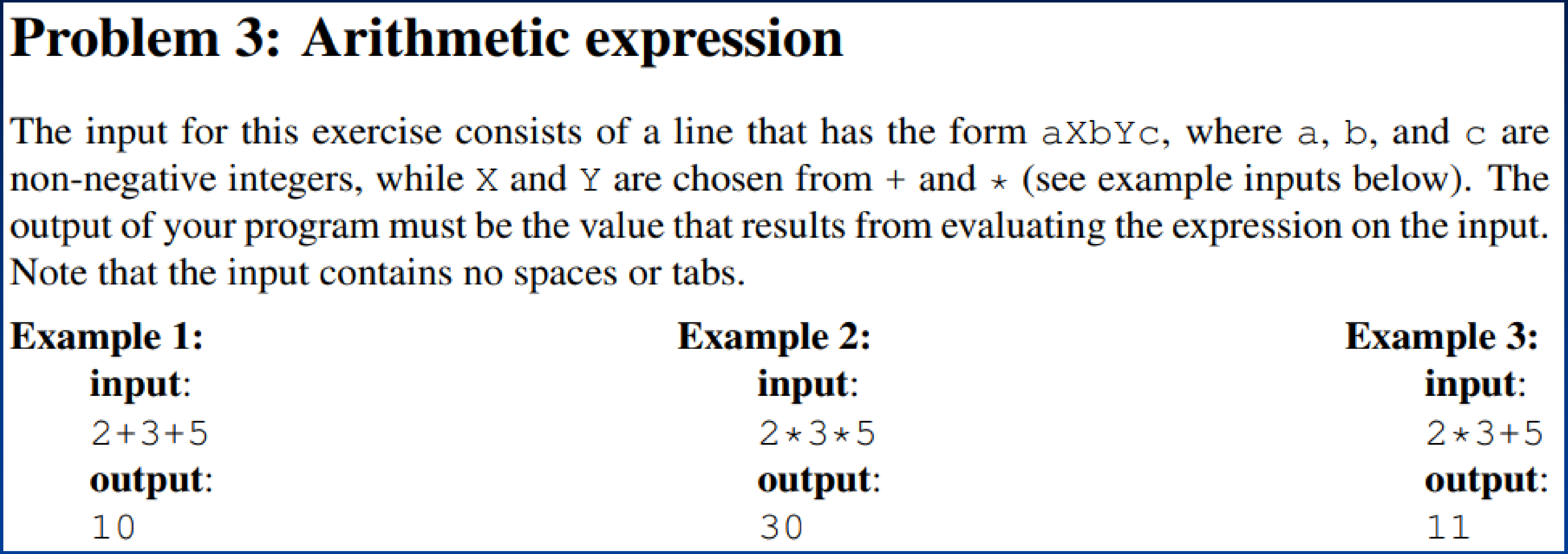
Example 1:
input:
```
240
```
output:
```
10
```

Example 2:
input:
```
480
```
output:
```
20
```

Example 3:
input:
```
2400
```
output:
```
100
```

```
int main(int argc, char *argv[]) {
  int n;
  scanf("%d", &n);
  printf("%d\n", (n + 23)/24);
  return 0;
}
```

Solutions 1st lab session



Problem 3: Arithmetic expression

The input for this exercise consists of a line that has the form `aXbYc`, where `a`, `b`, and `c` are non-negative integers, while `X` and `Y` are chosen from + and * (see example inputs below). The output of your program must be the value that results from evaluating the expression on the input. Note that the input contains no spaces or tabs.

Example 1:

input:
```
2+3+5
```
output:
```
10
```

Example 2:

input:
```
2*3*5
```
output:
```
30
```

Example 3:

input:
```
2*3+5
```
output:
```
11
```

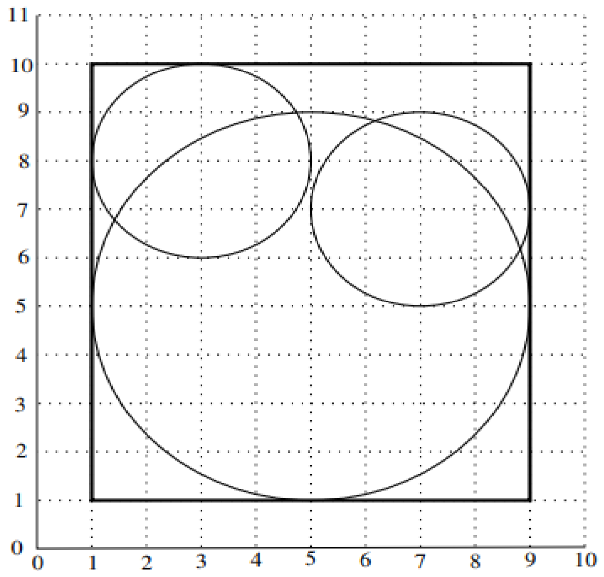
Solutions 1st lab session

```
int main(int argc, char *argv[]) {
  int a, b, c, e;
  char op1, op2;
  scanf("%d%c%d%c%d", &a, &op1, &b, &op2, &c);
  if (op1 == '+') {
    e = a + (op2 == '+' ? b+c : b*c);
  } else {
    a *= b;
    e = (op2 == '+' ? a+c : a*c);
  }
  printf("%d\n", e);
  return 0;
}
```

Solutions 1st lab session

Problem 4: Bounding Box

In the figure on the right, you see a grid containing three circles. The largest circle is defined by the center coordinate $(5, 5)$ and has radius 4. The smaller circles both have radius 2. The left one has its center at $(3, 8)$, while the right one has its center at $(7, 7)$. The rectangle is the smallest axes-aligned rectangle that surrounds the three circles completely. This rectangle is called the *bounding box* of the three circles. The bottom left corner point of this bounding box is the grid point $(1, 1)$, and the top right corner is the point $(9, 10)$.



Make a program that reads from the input the center points and radii of three circles. Per line there are three integers. The first two are the coordinates of the center point (x, y) of a circle. The third is its radius. The program must print on the screen the bottom left corner point and the top right corner point of the bounding box of the three circles. Make sure that your program produces output in the exact same format as given in the following examples.

Example 1 (see figure):

input:
```
5 5 4
3 8 2
7 7 2
```
output:
```
[(1,1),(9,10)]
```

Example 2:

input:
```
2 3 2
5 4 2
4 7 1
```
output:
```
[(0,1),(7,8)]
```

Example 3:

input:
```
5 5 5
4 4 4
3 3 3
```
output:
```
[(0,0),(10,10)]
```

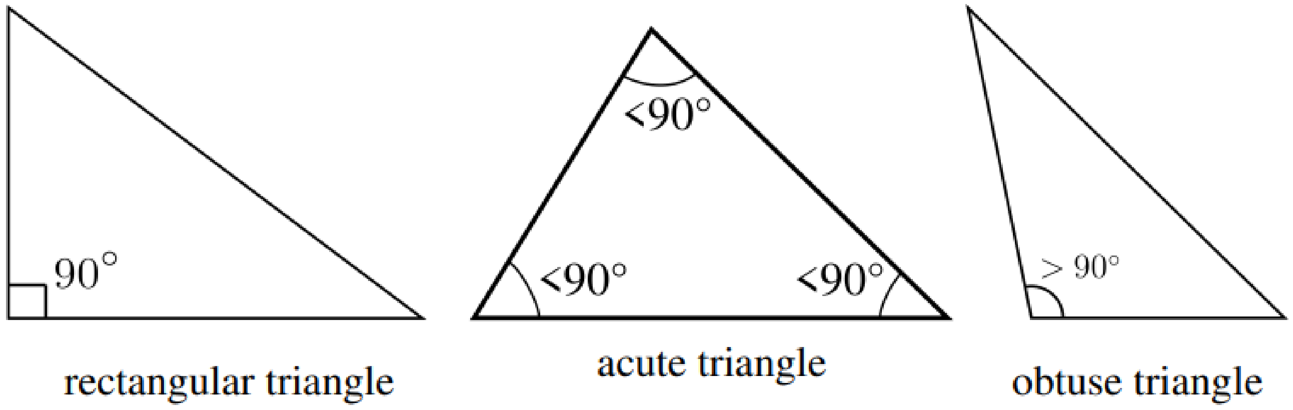
Solutions 1st lab session

```
int main(int argc, char *argv[]) {
  int x, y, r, minX, maxX, minY, maxY;
  scanf ("%d %d %d", &x, &y, &r);
  minX = x - r;
  maxX = x + r;
  minY = y - r;
  maxY = y + r;
  scanf ("%d %d %d", &x, &y, &r);
  minX = (x - r < minX ? x – r : minX);
  maxX = (x + r > maxX ? x + r : maxX);
  minY = (y - r < minY) ? y – r : minY);
  maxY = (y + r > maxY ? y + r : maxY);
  scanf ("%d %d %d", &x, &y, &r);
  minX = (x - r < minX ? x – r : minX);
  maxX = (x + r > maxX ? x + r : maxX);
  minY = (y - r < minY) ? y – r : minY);
  maxY = (y + r > maxY ? y + r : maxY);
  printf("[(%d,%d),(%d,%d)]\n", minX, minY, maxX, maxY);
  return 0;
}
```

Solutions 1st lab session

Problem 5: Triangles

Recall that a *rectangular triangle* is a triangle in which two sides are perpendicular. An *acute triangle* is a triangle with three acute angles (less than 90 degrees), while an *obtuse* triangle has one obtuse angle (greater than 90 degrees).



Make a program that reads from the input three positive integers a, b, and c. Each of these numbers is at most 1000. These numbers represent the lengths of the sides of a (possible) triangle. The output of your program must be `RECTANGULAR` if a rectangular triangle can be formed with these lengths. If an acute triangle can be formed, then the output must be `ACUTE`, and the output must be `OBTUSE` in case an obtuse triangle can be formed. The output must be `NONE` if none of these cases apply.

Example 1:

input:
```
3 4 5
```
output:
```
RECTANGULAR
```

Example 2:

input:
```
3 4 6
```
output:
```
OBTUSE
```

Example 3:

input:
```
3 4 4
```
output:
```
ACUTE
```

Solutions 1st lab session

```
int main(int argc, char *argv[]) {
  int a, b, c, h;
  scanf("%d%d%d", &a, &b, &c);
  if (b < a) {
    h = a;
    a = b;
    b = h;
  }
  if (c < b) {
    h = b;
    b = c;
    c = h;
  }
  // c == MAXIMUM(a,b,c)
  if (a + b <= c) {
     printf("NONE\n");
  } else {
     if (a*a + b*b == c*c) {
        printf("RECTANGULAR\n");
     } else {
       printf (a*a + b*b > c*c ? "ACUTE\n" : "OBTUSE\n");
     }
  }
  return 0;
}
```

A short introduction to modular arithmetic

Recall that for integers n and $d > 0$ we have:

$n = (n\ div\ d) * d + r$ and $0 \leq r < d$,

where $r = n\ mod\ d$.

Note that div is implemented in C as `/` (on integers) and mod as `%`.

Lemma 1: For modular addition we have

$(a + b) mod\ d = ((a\ mod\ d) + (b\ mod\ d)) mod\ d$

A short introduction to modular arithmetic

Lemma 1: For modular addition we have

$$(a + b) \bmod d = ((a \bmod d) + (b \bmod d)) \bmod d$$

Proof: Let $a = p * d + r$ and $b = q * d + s$, with $0 \leq r, s < d$. Then

- $\quad (a + b) \bmod d$
- $= \left((p + q) * d \; + \; r \; + \; s\right) \bmod d$
- $= (r \; + \; s) \bmod d$
- $= \left((p * d + r) \bmod d \; + \; (q * d + s) \bmod d\right) \bmod d$
- $= ((a\ mod\ d) + (b\ mod\ d))mod\ d$

A short introduction to modular arithmetic

Lemma 2: For modular multiplication we have

$$(a * b) \bmod d = ((a \bmod d) * (b \bmod d)) \bmod d$$

Proof: This is a direct consequence of lemma 1.

You only need to realize that $a * b = \sum_{i=1}^{b} a$ and apply lemma 1.

A short introduction to modular arithmetic

Lemma 3 For modular exponentiation we have

$$(a^n) \bmod d = ((a \bmod d)^n) \bmod d$$

Proof: This is a direct consequence of lemma 2.

You only need to realize that $a^n = \prod_{i=1}^{n} a$ and apply lemma 2.

Modular exponentiation

A simple modification of the fast version of the exponentiation algorithm can be used to compute the last 2 digits of x^n for very large values of n without overflow.

For example, take $x = 2$ and $n = 100$.

It is clearly silly to compute 2^{100} (it overflows, because a `long` can be at most 2^{63}-1), followed by taking the last 2 digits (i.e. modulo 100).

Modular exponentiation

Much better is:

$$2^{100} \bmod 100 = 4^{50} \bmod 100 = 16^{25} \bmod 100 =$$
$$\left(16 \times (16^{24} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (256^{12} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (56^{12} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (3136^{6} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (36^{6} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (1296^{3} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times (96^{3} \bmod 100)\right) \bmod 100 =$$
$$\left(16 \times 96 \times (96^{2} \bmod 100)\right) \bmod 100 =$$
$$\left(1536 \times (96^{2} \bmod 100)\right) \bmod 100 =$$
$$\left(36 \times (9216 \bmod 100)\right) \bmod 100 =$$
$$(36 \times 16) \bmod 100 = 576 \bmod 100 = 76$$

Try yourself to adapt the fast exponentiation algorithm such that it returns $a^b mod\ d$ instead of a^b.

Divisible by 11?

Claim: A number n is divisible by 11 if the alternating sum of its digits is divisible by 11.

2728 yields 2-7+2-8=-11 so 2728 is divisible by 11.

1728 yields 1-7+2-8=-12 so 1728 is not divisible by 11.

Why does this work?

Modular arithmetic is the key to the answer!

Divisible by 11?

First note that $10\ mod\ 11\ =\ (-1)\ mod\ 11$.

Using lemma 1: 10 mod 11 = (11 -1) mod 11 = (-1) mod 11

So, using lemma 3: $10^n\ mod\ 11 = (-1)^n\ mod\ 11$

Divisible by 11?

Using this fact, and lemmas 1 and 2 we get:

$$\left(\sum_{i=0}^{n} d_i \cdot 10^i\right) mod\ 11$$

$$= \left(\sum_{i=0}^{n} (d_i \cdot 10^i) mod\ 11\right) mod\ 11$$

$$= \left(\sum_{i=0}^{n} (-1)^i \cdot d_i\right) mod\ 11$$

QED.

Divisible by 13?

Claim:

A number $10 \cdot n + d$ (where $n > 0$ and $0 \leq d \leq 9$) is divisible by 13 if and only if $n + 4 \cdot d$ is divisible by 13.

169: $16 + 4 \cdot 9 = 52 = 4 \cdot 13$, so 169 is divisible by 13.

170: $17 + 4 \cdot 0 = 17$, so 170 is not divisible by 13.

Why does this work?

Modular arithmetic again is the key to the answer!

Divisible by 13?

A number $10 \cdot n + d$ (where $n > 0$ and $0 \leq d \leq 9$) is divisible by 13 if and only if $n + 4 \cdot d$ is divisible by 13.

Proof:

$$10 \cdot n + d = 10 \cdot n + 40 \cdot d \; - 40 \cdot d + d$$
$$= 10(n + 4 \cdot d) - 39d$$

39 is divisible by 13.

Clearly, 10 is not divisible by 13.

Since 13 is a prime number, we need $\mathrm{n} + 4 \cdot d$ to be divisible by 13 for $10 \cdot n + d$ to be divisible by 13.

Perfect numbers

A perfect number is a positive integer that is equal to the sum of its divisors (excluding the number itself).

For example, 6 has divisors 1, 2, and 3, and 6=1+2+3. So, 6 is a perfect number.

Write a program fragment that reads a positive integer n from the keyboard and outputs YES if n is a perfect number, and NO otherwise.

Perfect numbers

```
int n, sum=1, divisor=2;
scanf("%d", &n);
while (2*divisor <= n) {
  if (n%divisor == 0) {
    sum += divisor;
  }
  divisor++;
}
if (sum == n) {
  printf("%d\n", n);
}
```

Amicable numbers

Amicable numbers are two different positive integers related in such a way that the sum of the proper divisors of each is equal to the other number.

Example: 220 and 284 are amicable because the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110, of which the sum is 284; and the proper divisors of 284 are 1, 2, 4, 71 and 142, of which the sum is 220.

Amicable numbers

```
int m, n, msum=1, nsum=1, divisor=2;
scanf("%d %d", &m, &n);
int upb = (m > n ? m : n);
while (2*divisor <= upb) {
  msum += ((divisor < m) && m%divisor == 0) ? divisor : 0);
  nsum += ((divisor < n) && n%divisor == 0) ? divisor : 0);
   divisor++;
}
if ((msum == n) && (nsum == m)) {
  printf("YES\n");
} else {
  printf("NO\n");
}
```

Sum of all divisors from 1 to n

Given an integer n (where n > 0).

Determine the sum of the divisors of the numbers 1, 2, 3, .., n.

Example: n=4

1: 1

2: 1 + 2

3: 1 + 3

4: 1 + 2 + 4

Sum = 1 + (1 + 2) + (1 + 3) + (1 + 2 + 4)=15

Sum of all divisors from 1 to n

Given an integer n (where n > 0).

Determine the sum of the divisors of the numbers 1, 2, 3, .., n.

Example: n=4

1: 1

2: 1 + 2

3: 1 + 3

4: 1 + 2 + 4

Sum = 1 + (1 + 2) + (1 + 3) + (1 + 2 + 4)=15

Sum of all divisors from 1 to n

Maybe you are tempted to write a program of this form.

```
sum = 0;
for (int i=1; i <= n; i++) {
    sum += sum of divisors of i;
}
```

Sum of all divisors from 1 to n

```
n=9
1: 1
2: 1 + 2
3: 1 +     3
4: 1 + 2 +   4
5: 1 +          + 5
6: 1 + 2 + 3         + 6
7: 1 +                    7
8: 1 + 2 +   4 +            + 8
9: 1 +     3                    + 9
```

Sum of all divisors from 1 to n

If you see the structure (using the previous slide) then you realize that the following efficient code solves the problem.

```
int sum = 0;
for (d=1; d <= n; d++) {
   sum += (n/d)*d;
}
```

GCD: Greatest Common Divisor

- Let a and b be non-negative integers.
- GCD(a,b) is the greatest integer that divides a and b without a remainder.
  - In other words, it is the largest common factor of a and b.

- **Euclid of Alexandria** was a Greek mathematician from the 3rd century before Christ. He is the inventor of the *Euclidean algorithm* for finding the greatest common divisor of two non-negative integers.

GCD: Euclid's algorithm

It is clear that gcd(a, b) = gcd(b, a) and gcd(a, 0) = gcd(0, a) = a.

For natural numbers a, b (where a > b) the following theorem holds:

$$\gcd(a, b) = \gcd(a-b, b)$$

Example: gcd(34, 8) = gcd(26, 8) = gcd(18, 8) = gcd(10, 8) = gcd(2, 8)
gcd(8, 2) = gcd(6, 2) = gcd(4, 2) = gcd(2, 2) = gcd(0, 2)=2

GCD: Euclid’s algorithm

Of course, an example is not a proof.

We need to prove that (for a>b):

gcd(a, b) = gcd(a-b, b)

Let a>b and gcd(a, b) = c.

First we show that c is a divisor of a-b and b.

gcd(a, b) = c, so c is a divisor of a and b.

Hence, there exist natural numbers m and n such that:
a = m*c and b = n*c.

So, a - b = (m - n)*c, and c is a divisor of a – b.

GCD: Euclid's algorithm

Next, we prove that c is the *greatest* divisor of b and a - b.

- Let d>c be a divisor of b and a - b.
- Hence, there exist natural numbers p and q such that: a - b = p*d and b = q*d.
- So, a = p*d + b = (p + q)*d.
- Hence, d is a divisor of a and b.
- But, this contradicts that gcd(a , b) = c, since this would mean that d is greater than gcd(a,b).
- So, we conclude that c is the greatest divisor of b and a - b. This is called a *proof by contradiction*.

Logic: Proof by contradiction

In the logic course you learn that:

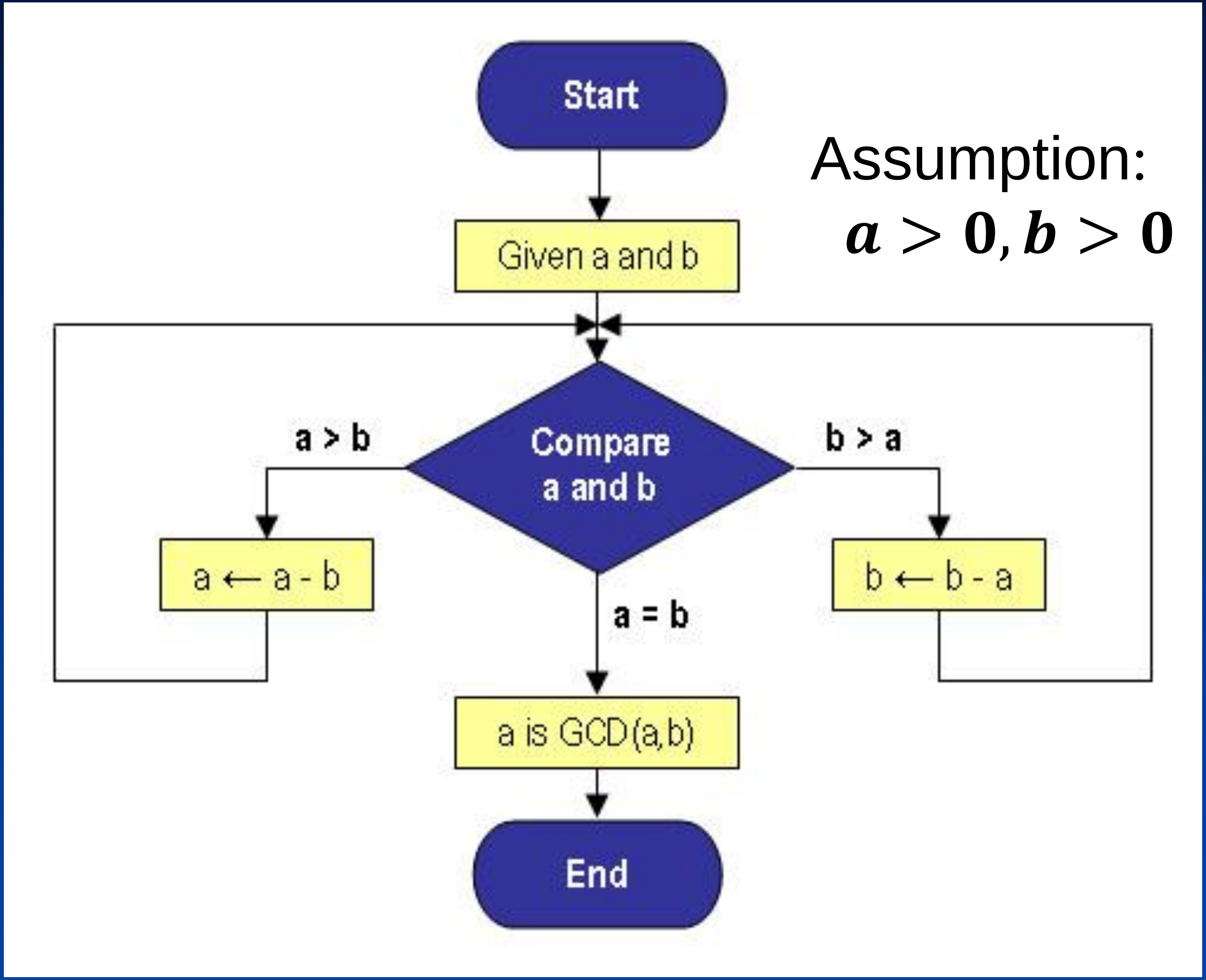
$P \Rightarrow Q$ is equivalent with $\neg P \vee Q$

A consequence of this is that:

$P \Rightarrow \text{false}$ is equivalent with $\neg P \vee \text{false} = \neg P$

So, if we assume P and infer false, then we have a proof of $\neg P$.

GCD: Euclid’s algorithm

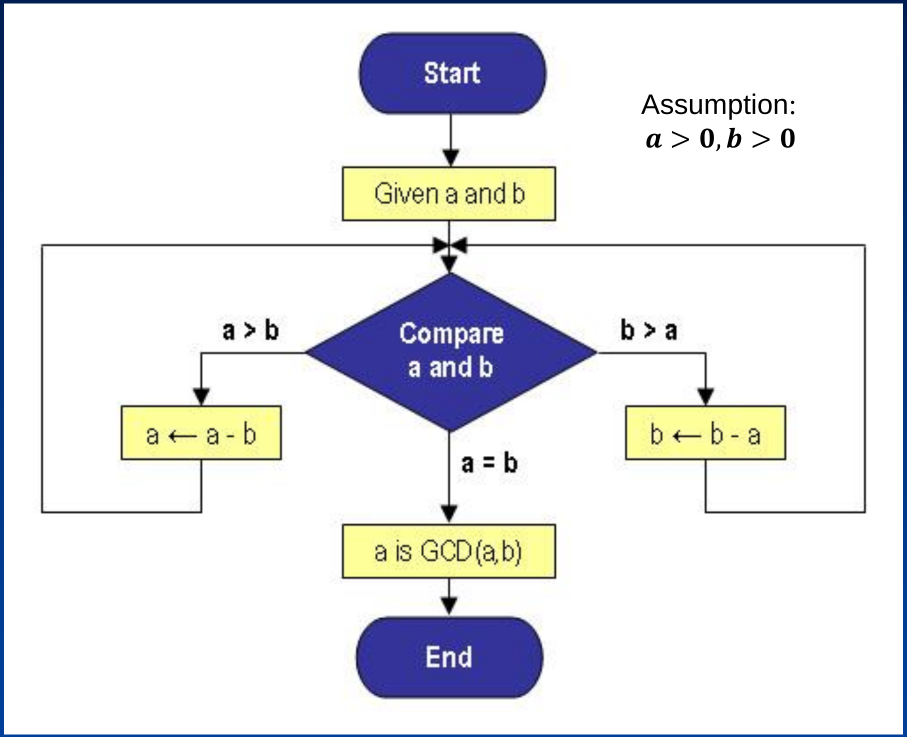


Assumption:
$a > 0, b > 0$

gcd(a, b) = gcd(a-b, b)

GCD: Euclid’s algorithm

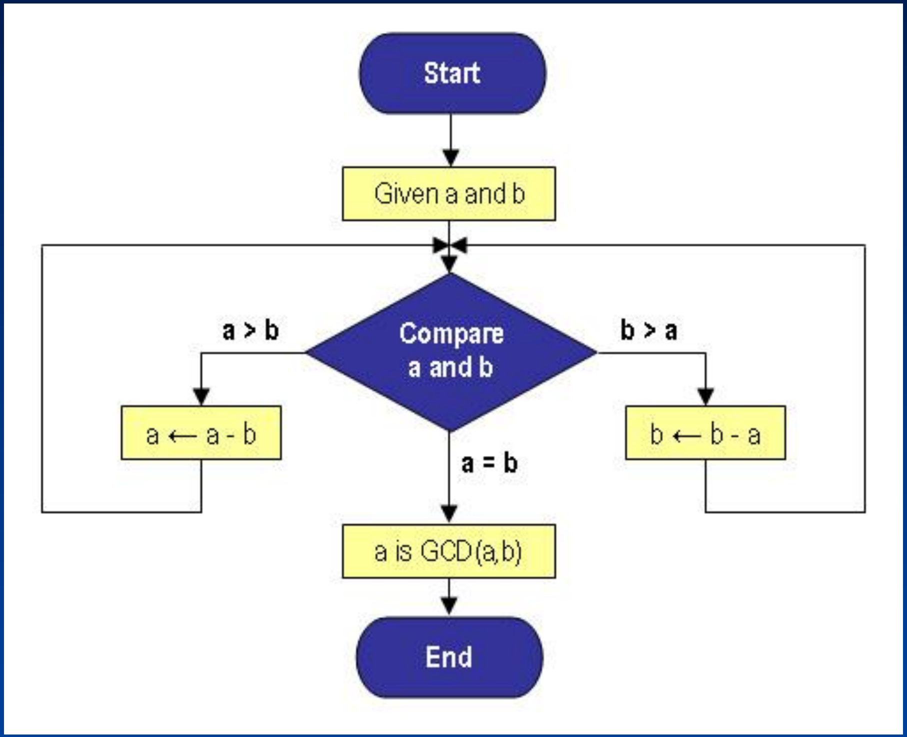
```
/* a==A, b==B */
if ((a == 0) || (b == 0)) {
  a = a + b;
} else {
  while (a != b) {
    if (a > b) {
      a = a- b;
    } else {
      b = b - a;
    }
  }
  /* a == b == gcd(A,B) */
}
/* a == gcd(A,B) */
```



Assumption:
$a > 0, b > 0$

GCD: Euclid's algorithm

```
/* a==A, b==B */
if ((a == 0) || (b == 0)) {
  a = a + b;
} else {
  while (a != b) {
    while (a > b) {
      a = a - b;
    }
    while (b > a) {
      b = b - a;
    }
  }
  /* a == b == gcd(A,B) */
}
/* a == gcd(A,B) */
```



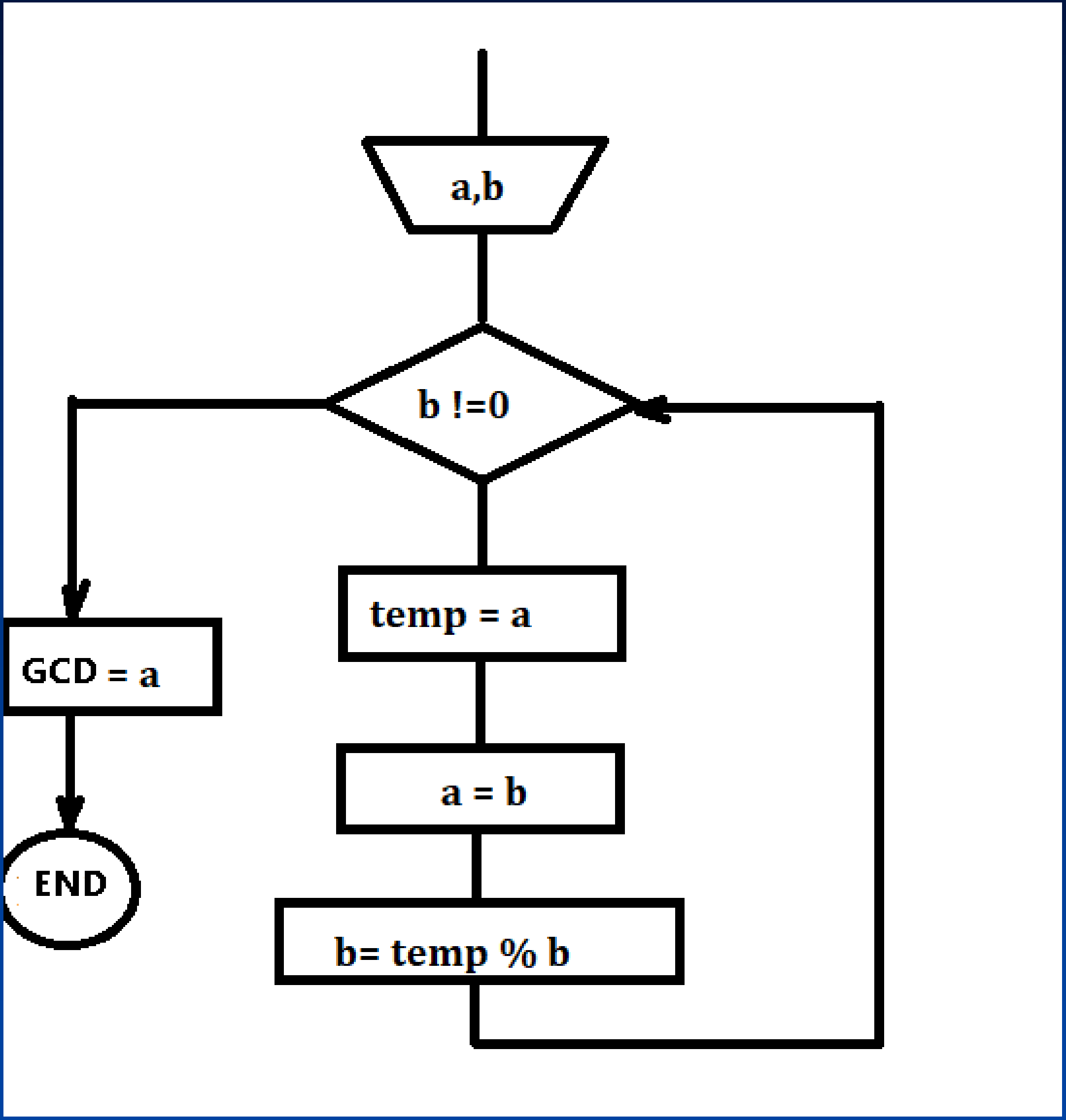
GCD: Euclid's algorithm

```
/* a==A, b==B */
if ((a == 0) || (b == 0)) {
  a = a + b;
} else {
  while (a != b) {
    while (a >= b) {
      a = a - b;
    }
    while (b >= a) {
      b = b - a;
    }
  }
  /* a == b == gcd(A,B) */
}
/* a == gcd(A,B) */
```

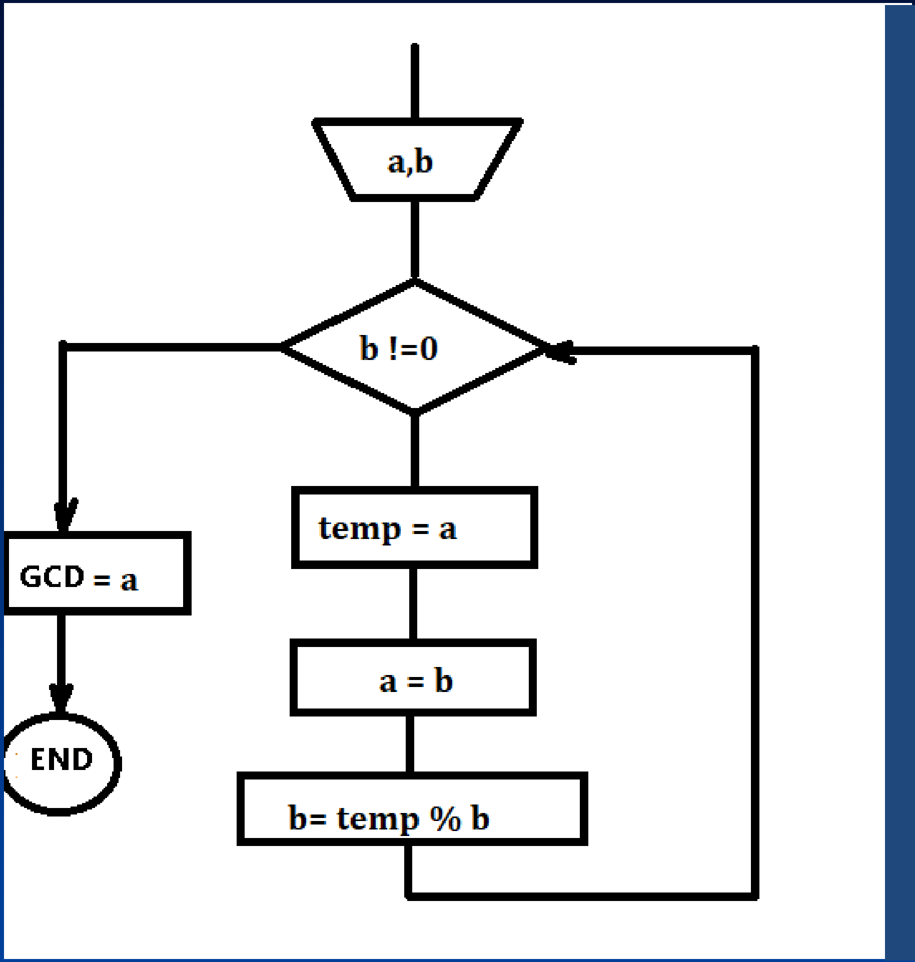
Why is this version of the algorithm wrong?

Well, after termination of the loop with the guard **`a>=b`** we could have **`a == 0`**. The next loop will not terminate in that case!

GCD: a bit smarter



GCD: Euclid's algorithm



```
/* a==A, b==B */
while (b != 0) {
   int r = a%b;
   a = b;
   b = r;
}
/* a==gcd(A,B) */
```

But how can we be sure that this algorithm terminates?
Well, it terminates because **b** is a positive integer, and it decreases in each iteration. Hence, it must eventually reach 0.

GCD: application

Write a program that simplifies a fraction a/b that is read from the keyboard.

So, for input a/b=45/189 it should output 5/21.

Solution sketch:

1. Read a and b from the keyboard
2. Compute g=gcd(a,b)
3. numerator=a/g
4. denominator = b/g
5. Output numerator “/” denominator

Nested loops

The body of loop can contain any kind of statement in its body, including another loop.

To avoid a conflict with the outer loop, the inner loop must have a different name for its loop variable than the outer loop.

```
for (int i = 0; i < 3; i++) {
  for (int j = 1; j <= 2; j++) {
    printf ("i=%d, j=%d\n", i, j);
  }
}
```

```
i=0, j=1
i=0, j=2
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
```

Nested for loop example

What is the output of the following nested for loop?

```
for (int i = 1; i <= 5; i++) {
  for (int j = 0; j < i; j++) {
    putchar('*');
  }
  putchar('\n');
}
```

```
*
**
***
****
*****
```

Nested for loop example

What is the output of the following nested for loop?

```
for (int i = 1; i <= 5; i++) {
  for (int j = 0; j < i; j++) {
    printf("%d", i);
  }
  putchar('\n');
}
```

```
1
22
333
4444
55555
```

Nested for loop example

What is the output of the following nested for loop?

```
for (int i = 1; i <= 5; i++) {
  for (int j = 0; j < i; j++) {
    printf("%d", j);
  }
  putchar('\n');
}
```

```
0
01
012
0123
01234
```

Nested loops: Counting primes

Exercise: write a program that reads a positive integer **n** from the keyboard and prints the number of primes that are less than **n**.

```
int n, cnt = 0;
scanf ("%d", &n);
while (n > 0) {
  n--;
  int a = 3, sq = 9;
  while (sq <= n) {
    if (n%a == 0) {
      break; // n is not a prime
    }
    sq += 4*a + 4;
    a += 2;
  } // inner while
  cnt += ((n==2) || ((n >= 2) && (n%2) && (sq > n));
} // outer while
printf("%d primes\n", cnt);
```

End week 3