Introduction to Programming (in C++)

Loops

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

Example

- Assume the following specification:

  Input: read a number N > 0
  Output: write the sequence 1 2 3 … N
  (one number per line)

- This specification suggests some algorithm with a *repetitive* procedure.

The *while* statement

- Syntax:

  while (⟨condition⟩) statement;

  (the condition must return a Boolean value)

- Semantics:
  - Similar to the repetition of an *if* statement
  - The condition is evaluated:
    - If *true*, the statement is executed and the control returns to the while statement again.
    - If *false*, the while statement terminates.

Write the numbers 1…N

```
// Input:  read a number N > 0
// Output: write the numbers 1...N
           (one per line)

int main() {
    int N;
    cin >> N;
    int i = 1;
    while (i <= N) {
       // The numbers 1..i-1 have been written
        cout << i << endl;
        i = i + 1;
    }
}
```

Product of two numbers

```
//Input:  read two non-negative numbers x and y
//Output: write the product x*y

// Constraint: do not use the * operator

// The algorithm calculates the sum x+x+x+…+x (y times)

int main() {
    int x, y;
    cin >> x >> y;  // Let x=A, y=B
    int p = 0;
    // Invariant: A*B = p + x*y
    while (y > 0) {
        p = p + x;
        y = y - 1;
    }
    cout << p << endl;
}
```

A quick algorithm for the product

- Let p be the product $x * y$
- Observation
  - If y is even, $p = (x * 2) * (y/2)$
  - If y is odd, $p = x * (y\text{-}1) + x$ and (y-1) becomes even
- Example: $17 * 38 = 646$

x	y	Δp
17	38	
34	19	
34	18	34
68	9	
68	8	68
136	4	
272	2	
544	1	
544	0	544
		646

A quick algorithm for the product

```
int main() {
    int x, y;
    cin >> x >> y; // Let x=A, y=B
    int p = 0;
    // Invariant: A*B = p + x*y
    while (y > 0) {
        if (y%2 == 0) {
            x = x*2;
            y = y/2;
        }
        else {
            p = p + x;
            y = y - 1;
        }
    }
    cout << p << endl;
}
```

x	y	p
17	38	0
34	**19**	0
34	18	34
68	**9**	34
68	8	102
136	4	102
272	2	102
544	**1**	102
544	0	**646**

Why is the quick product interesting?

- Most computers have a multiply instruction in their machine language.

- The operations $x*2$ and $y/2$ can be implemented as 1-bit left and right shifts, respectively. So, the multiplication can be implemented with shift and add operations.

- The quick product algorithm is the basis for hardware implementations of multipliers and mimics the paper-and-pencil method learned at school (but using base 2).

Quick product in binary: example

```
77 x 41 = 3157

      1001101
    x 0101001
      -------
      1001101
   1001101
 1001101
 ------------
 110001010101
```

Counting a's

- We want to read a text represented as a sequence of characters that ends with '.'
- We want to calculate the number of occurrences of the letter 'a'
- We can assume that the text always has at least one character (the last '.')
- Example: the text

  Programming in C++ is amazingly easy!.

  has 4 a's

Counting a’s

```
// Input:  sequence of characters that ends with ‘.’
// Output: number of times ‘a’ appears in the
//         sequence

int main() {
    char c;
    cin >> c;
    int count = 0;
    // Inv: count is the number of a’s in the visited
    //      prefix of the sequence. c contains the next
    //      non-visited character
    while (c != ‘.’) {
        if (c == ‘a’) count = count + 1;
        cin >> c;
    }

    cout << count << endl;
}
```

Counting digits

- We want to read a non-negative integer and count the number of digits (in radix 10) in its textual representation.

- Examples

  8713105 → 7 digits
  156 → 3 digits
  8 → 1 digit
  0 → 1 digit (note this special case)

Counting digits

```
// Input:  a non-negative number N
// Output: number of digits in N (0 has 1 digit)

int main() {
    int N;
    cin >> N;
    int ndigits = 0;

    // Inv: ndigits contains the number of digits in the
    //      tail of the number, N contains the remaining
    //      part (head) of the number
    while (N > 9) {
        ndigits = ndigits + 1;
        N = N/10;  // extracts one digit
    }

    cout << ndigits + 1 << endl;
}
```

Euclid’s algorithm for gcd

- Properties
  - gcd(a,a)=a
  - If a > b, then gcd(a,b) = gcd(a-b,b)
- Example

a	b
114	42
72	42
30	42
30	12
18	12
6	12
6	6

gcd (114, 42)

Euclid’s algorithm for gcd

```
// Input:  read two positive numbers (a and b)
// Output: write gcd(a,b)

int main() {
    int a, b;
    cin >> a >> b; // Let a=A, b=B
    // gcd(A,B) = gcd(a,b)
    while (a != b) {
        if (a > b) a = a – b;
        else b = b – a;
    }
    cout << a << endl;
}
```

Faster Euclid's algorithm for gcd

- Properties
  - $gcd(a, 0)=a$
  - If $b > 0$ then $gcd(a, b) = gcd(b, a \bmod b)$
- Example

a	b
114	42
42	30
30	12
12	6
6	0

Faster Euclid's algorithm for gcd

```
// Input:  read two positive numbers (a and b)
// Output: write gcd(a,b)

int main() {
    int a, b;
    cin >> a >> b; // Let a=A, b=B
    // gcd(A,B) = gcd(a,b)
    while (b != 0) {
        int r = a%b;
        a = b;
        b = r;  // Guarantees b < a (loop termination)
    }
    cout << a << endl;
}
```

Efficiency of Euclid's algorithm

- How many iterations will Euclid's algorithm need to calculate gcd(a,b) in the worst case (assume a > b)?
  - Subtraction version: a iterations (consider gcd(1000,1))
  - Modulo version: $\leq 5 * d(b)$ iterations, where d(b) is the number of digits of b represented in base 10 (proof by Gabriel Lamé, 1844)

Solving a problem several times

- In many cases, we might be interested in solving the same problem for several input data.

- Example: calculate the gcd of several pairs of natural numbers.

Input	Output
12 56 30 30 1024 896 100 99 17 51	4 30 128 1 17

Solving a problem several times

```
// Input:  several pairs of natural numbers at the input
// Output: the gcd of each pair of numbers written at the output

int main() {
    int a, b;
    // Inv: the gcd of all previous pairs have been
    //      calculated and written at the output
    while (cin >> a >> b) {
        // A new pair of numbers from the input
```

Calculate gcd(a,b) and
write the result into cout

```
    }
}
```

Solving a problem several times

```
// Input:  several pairs of natural numbers at the input
// Output: the gcd of each pair of numbers written at the output

int main() {
    int a, b;
    // Inv: the gcd of all previous pairs have been
    //      calculated and written at the output
    while (cin >> a >> b) {
        // A new pair of numbers from the input
        while (b != 0) {
            int r = a%b;
            a = b;
            b = r;
        }
        cout << a << endl;
    }
}
```

Prime number

- A **prime number** is a natural number that has exactly two *distinct* divisors: 1 and itself. (Comment: 1 is not prime)

- Write a program that reads a natural number (N) and tells whether it is prime or not.

- Algorithm: try all potential divisors from 2 to N-1 and check whether the remainder is zero.

Prime number

```
// Input:  read a natural number N>0
// Output: write “is prime” or “is not prime” depending on
//         the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);
    // 1 is not prime, 2 is prime, the rest enter the loop (assume prime)

    // is_prime is true while a divisor is not found
    // and becomes false as soon as the first divisor is found
    while (divisor < N) {
        if (N%divisor == 0) is_prime = false;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
```

Prime number

- Observation: as soon as a divisor is found, there is no need to check divisibility with the rest of the divisors.

- However, the algorithm tries all potential divisors from 2 to N-1.

- Improvement: stop the iteration when a divisor is found.

Prime number

```
// Input:  read a natural number N>0
// Output: write “is prime” or “is not prime” depending on
//         the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);

    while (is_prime and divisor < N) {
        is_prime = N%divisor != 0;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
```

Prime number: doing it faster

- If N is not prime, we can find two numbers, *a* and *b*, such that:

  $$N = a * b, \qquad \text{with } 1 < a \le b < N$$

  and with the following property: $a \le \sqrt{N}$

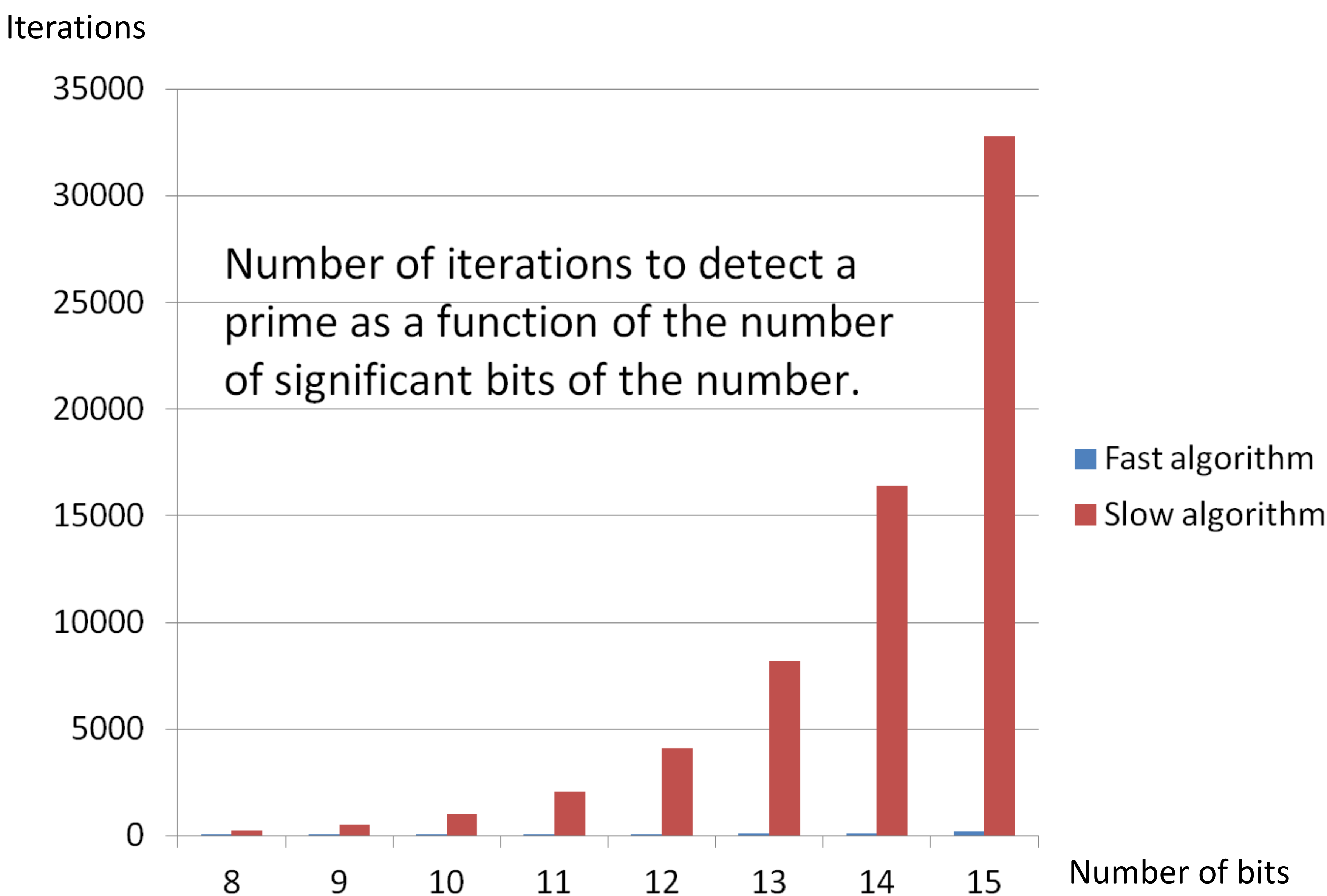
- There is no need to find divisors up to N-1. We can stop much earlier.

- Note: $a \le \sqrt{N}$ is equivalent to $a^2 \le N$

Prime number: doing it faster

```
// Input:  read a natural number N>0
// Output: write “is prime” or “is not prime” depending on
//         the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);

    while (is_prime and divisor*divisor <= N) {
        is_prime = N%divisor != 0;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
```

Is there any real difference?



In real time (N= 2110454939)

```
> time prime_slow < number
is prime
10.984u 0.004s 0:11.10 98.9%
```

```
> time prime_fast < number
is prime
0.004u 0.000s 0:00.00 0.0%
```

The *for* statement

- Very often we encounter loops of the form:

```
i = N;
while (i <= M) {
    do_something;
    i = i + 1;
}
```

- This can be rewritten as:

```
for (i = N; i <= M; i = i + 1) {
    do_something;
}
```

The *for* statement

- In general

```
for (⟨S_init⟩; ⟨condition⟩; ⟨S_iter⟩)  ⟨S_body⟩;
```

is equivalent to:

```
S_init;
while (⟨condition⟩) {
   ⟨S_body⟩;
   ⟨S_iter⟩;
}
```

Writing the numbers in an interval

```
// Input:  read two integer numbers, N and M,
//         such that N <= M.
// Output: write all the integer numbers in the
//         interval [N,M]

int main() {
    int N, M;
    cin >> N >> M;

    for (int i = N; i <= M; ++i) cout << i << endl;
}
```

Variable declared within the scope of the loop

Autoincrement operator

Calculate x^y

```
// Input:  read two integer numbers,
           x and y, such that y >= 0
// Output: write xʸ

int main() {
    int x, y;
    cin >> x >> y;
    int p = 1;
    for (int i = 0; i < y; ++i) p = p*x;
    cout << p << endl;
}
```

Drawing a triangle

- Given a number n (e.g. n = 6), we want to draw this triangle:

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

Drawing a triangle

```
// Input:  read a number n > 0
// Output: write a triangle of size n

int main() {
    int n;
    cin >> n;
    // Inv: the rows 1..i-1 have been written
    for (int i = 1; i <= n; ++i) {
        // Inv: '*' written j-1 times in row i
        for (int j = 1; j <= i; ++j) cout << '*';
        cout << endl;
    }
}
```

Perfect numbers

- A number $n > 0$ is perfect if it is equal to the sum of all its divisors except itself.

- Examples
  - 6 is a perfect number ($1+2+3 = 6$)
  - 12 is not a perfect number ($1+2+3+4+6 \neq 12$)

- Strategy
  - Keep adding divisors until the sum exceeds the number or there are no more divisors.

Perfect numbers

```
// Input:  read a number n > 0
// Output: write a message indicating
//         whether it is perfect or not

int main() {
    int n;
    cin >> n;

    int sum = 0, d = 1;
    // Inv: sum is the sum of all divisors until d-1
    while (d <= n/2 and sum <= n) {
        if (n%d == 0) sum += d;
        d = d + 1;
    }

    if (sum == n) cout << "is perfect" << endl;
    else cout << "is not perfect" << endl;
}
```

Perfect numbers

- Would the program work using the following loop condition?

```
while (d <= n/2 and sum < n)
```

- Can we design a more efficient version without checking all the divisors until n/2?
  – Clue: consider the most efficient version of the program to check whether a number is prime.