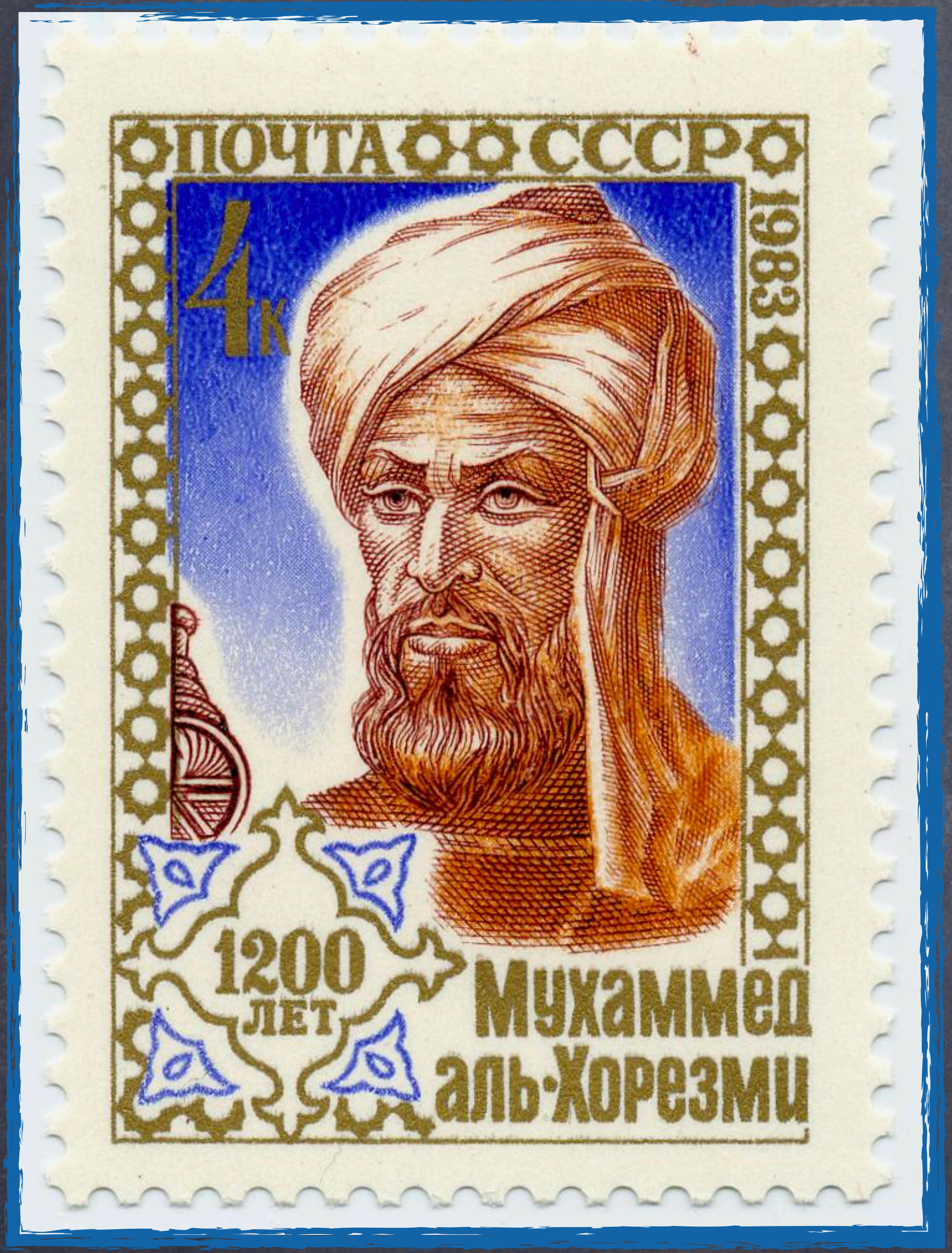
Algorithms without a computer

Cécile Mailler and Sarah Penington
(University of Bath)

What is an algorithm?



Muhammad Ibn Mūsā al-Khuwārizmī (800 AD)

input

↓

Sequence of operations

↓

output

What is an algorithm?

input	ingredients	meter reading
↓	↓	↓
Sequence of operations	Recipe	The gas company pricing formula
↓	↓	↓
output	cake	bill

What is an algorithm?

input

↓

Sequence of operations

↓

output

integer n

↓

Sequence of operations

↓

"yes" if n is divisible by 37
"no" otherwise

What is an algorithm?

n is divisible by 37
if there exists an integer k
such that

$n = 37 \times k$
$= 37 + \ldots + 37 \quad$ (k times)

integer n

↓

Sequence of
operations

↓

s divisible by 37
otherwise

To see whether n is divisible by 37, we can:
subtract 37 from it several times until what remains is less than 37, and look at what remains:

- if nothing remains, then "yes"
- otherwise, "no" ($n = 37 + \ldots + 37 + \text{sthg}$)

The algorithm Div37

Name of the algorithm and input(s)

```
Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
      aux <- aux-37

if aux = 0 then:
      return "yes"
else:
      return "no"
```

The algorithm Div37

```
Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"
```

Name of the algorithm and input(s)

Output

The algorithm Div37

```
Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"
```

Name of the algorithm and input(s)

Output

Only use basic operations:

basic calculus: + - x

tests: "aux is at least 37"
"aux=0"

assignations: aux <- n

The algorithm Div37

Algorithm Div37(n):

Name of the algorithm and input(s)

```
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"
```

Output

aux <- n
creates a memory cell and puts the value n in it

aux <- aux - 37
takes the current value in the cell "aux" removes 37 from it and places this new value in "aux".

The algorithm Div37

Algorithm Div37(n):

aux <- n

While aux is at least 37 do:

    aux <- aux-37

if aux = 0 then:

    return "yes"

else:

    return "no"

Name of the algorithm and input(s)

Output

if "test" then:

    "action 1"

else:

    "action 2"

If the "test" is true then execute "action 1", otherwise execute "action 2".

The algorithm Div37

Algorithm Div37(n):

aux <- n

While aux is at least 37 do:

    aux <- aux-37

if aux = 0 then:

    return "yes"

else:

    return "no"

Name of the algorithm and input(s)

Output

while "test" do:

    "action"

next instruction

1- If "test" is true then execute "action" and do 1 again.

2- If "test" is false, then execute "next instruction".

The algorithm Div37

Algorithm Div37(n):
```
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"
```

Execute Algorithm Div37(112):

The algorithm Div37

Algorithm Div37(n):

aux <- n

While aux is at least 37 do:

    aux <- aux-37

if aux = 0 then:

    return "yes"

else:

    return "no"

Execute Algorithm Div37(112):

aux

$\boxed{112}$

The algorithm Div37

```
Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
      aux <- aux-37

if aux = 0 then:
      return "yes"
else:
      return "no"
```

Execute Algorithm Div37(112):

aux
112

aux 75 | aux 38 | aux 1

The algorithm Div37

Algorithm Div37(n):

aux <- n

While aux is at least 37 do:

    aux <- aux-37

if aux = 0 then:

    return "yes"

else:

    return "no"

Execute Algorithm Div37(112):

aux $\boxed{112}$

aux $\boxed{75}$ aux $\boxed{38}$ aux $\boxed{1}$

return "no"

Complexity of an algorithm

```
Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
      aux <- aux-37

if aux = 0 then:
      return "yes"
else:
      return "no"
```

How efficient is this algorithm?

One possible measure is the "complexity" = number of elementary operations (+, -, x, assignations, tests).

The complexity is a function of the input.

Complexity of an algorithm

Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"

Execute Algorithm Div37(112):

aux: 112 +1 (assignation)

aux: 75, aux: 38, aux: 1 +1 (last test)

+3 +3 +3 (test+ subtraction + assignation)

return "no" +1 (test)

Complexity = 12 if n = 112

Complexity of an algorithm

Algorithm Div37(n):
aux <- n

While aux is at least 37 do:
    aux <- aux-37

if aux = 0 then:
    return "yes"
else:
    return "no"

Execute Algorithm Div37(n):

aux $\boxed{n}$ +1 (assignation)

aux $\boxed{n-37}$ aux $\boxed{n-72}$ aux $\boxed{n-109}$ ...

+3 times n/37 +1 (last test)

return "no" +1 (test)

$$\boxed{\begin{array}{l}\text{Complexity}\\ = 3 + 3 \times n/37\\ = 3 \times (1 + n/37)\end{array}}$$

Exercise 1

1- Write the algorithm Div11(n) that outputs "yes" if n is divisible by 11 and "no" otherwise. What is the complexity of Div11(n)?

2- Did you know that a number is divisible by 11 if and only if the alternating sum of its digits is divisible by 11?

For example, 41527 is not divisible by 11 because 4-1+5-2+7 = 13, but 50457 is divisible by 11 because 5-0+4-5+7 = 11.

Knowing this, check whether 145 379 is divisible by 11. How many elementary operations have you executed? How many would Div11(145 379) would execute to get the same answer?

Exercise 2

Write an algorithm Div(n,k) that takes as an input two integers n and k and gives the result of the division of n by k and its remainder term.

For example,

- Div(17,5) gives (3,2) as an output because 17 = 3 x 5 + 2.
- Div(3,5) gives (0,3) because 3 = 0 x 5 + 3
- Div(18,3) gives (6,0) because 18 = 6 x 3 + 0

What is the complexity of this algorithm as a function of n and k?

Finding the minimum

7 integers (n1,...,n7) → Min7 → their minimum (and its index)

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

↓

n5

3

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
min <- 12
index <- 1
```

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
        if ni < min then:
                min <- ni
                index <- i
return min and index
```

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
min <- 12
index <- 1
```

(i = 2)

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
        if ni < min then:
                min <- ni
                index <- i
return min and index
```

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
min <- 12
index <- 1
```

(i = 3)

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
        if ni < min then:
                min <- ni
                index <- i
return min and index
```

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

min <- 8
index <- 3

(i = 3)

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
        if ni < min then:
                min <- ni
                index <- i
return min and index
```

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
      if ni < min then:
            min <- ni
            index <- i
return min and index
```

```
min <- 8
index <- 3
```

(i = 4) (i = 5)

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
Min7(n1,...,n7):
min <- n1
index <- 1
For i = 2 to 7 do:
      if ni < min then:
            min <- ni
            index <- i
return min and index
```

min <- 3
index <- 5

(i = 5) (i = 6) (i = 7)

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
Min7(n1,...,n7):
min <- n1               +1
index <- 1              +1
For i = 2 to 7 do:
      if ni < min then: +1  |
          min <- ni     +1  | x6
          index <- i    +1  |
return min and index
```

The complexity of this algorithm is at most:

$$2+6\times 3 = 7\times 3-1 = 20$$

(Finding the minimum out of a list of n integers would have complexity ???)

Finding the minimum

n1	n2	n3	n4	n5	n6	n7
12	47	8	22	3	10	88

```
Min7(n1,...,n7):
min <- n1   +1
index <- 1  +1
For i = 2 to 7 do:
      if ni < min then: +1
            min <- ni +1         x6
            index <- i +1
return min and index
```

The complexity of this algorithm is:

$2+6x3 = 7x3-1 = 20$

(Finding the minimum out of a list of n integers would have complexity 3xn-1)

Sorting a list of numbers

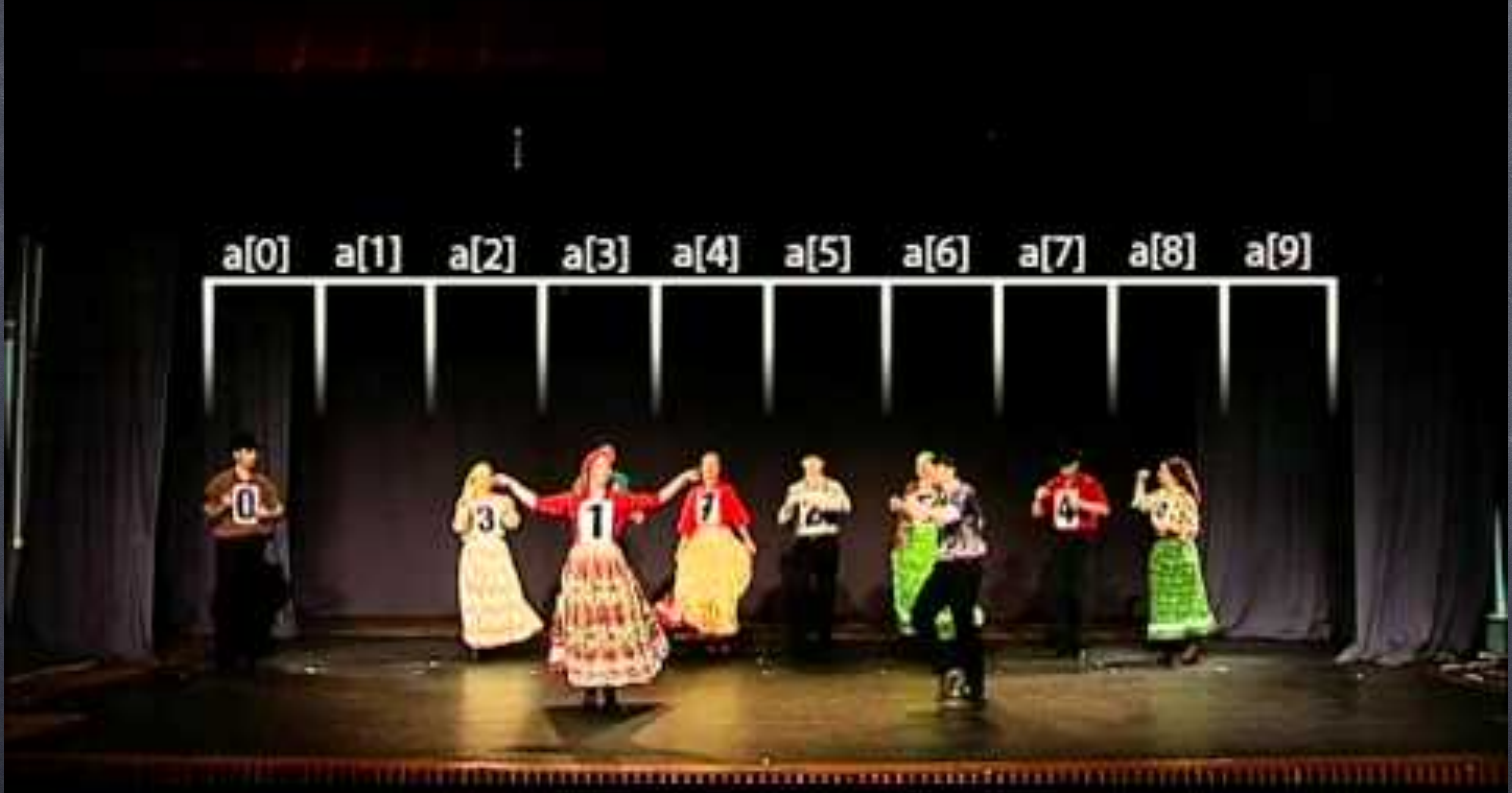
7 integers → Sort7 → the 7 integers sorted in increasing order

12	47	8	22	3	10	88

↓

3	8	10	12	22	47	88

SelectSort



Sorting a list of numbers

7 integers → Sort7 → the 7 integers sorted in increasing order

```
Sort7(n1,...,n7):
For i = 1 to 7 do:
     min, k <- Min(ni,..., n7)
     ni <-> nk ("exchange ni and nk")
return (n1,...,n7)
```

Sorting a list of numbers

7 integers → **Sort7** → the 7 integers sorted in increasing order

Complexity

$= 7 + 3 \times (7 + 6 + \ldots + 1) = 91$

To sort n integers:
complexity cst x n x n

```
Sort7(n1,...,n7):
For i = 1 to 7 do:
    min, k <- Min(ni,..., n7) + 3 x (8 - i)
    ni <-> nk ("exchange ni and nk")     +1
return (n1,...,n7)
```

Sorting a list of numbers

This algorithm is called SelectSort

Sorting is used every day (think of big data!): people have worked a lot on writing efficient algorithms for it.

One of the best algorithms is QuickSort, for which it is best to first shuffle your list of integers at random!
(to avoid rare bad configurations)

RSA algorithm

Message $\xrightarrow{\text{Public key}}$ Encrypted message

Encrypted message $\xrightarrow{\text{Private key}}$ Message

RSA algorithm

Public key

Message $\longrightarrow$ Encrypted message

Private key

Encrypted message $\longrightarrow$ Message

RSA stands for Rivest-Shamir-Adleman (invented the algorithm in 1978)

Clifford Cox (GCHQ) developed a similar algorithm in 1973 (declassified in 1997)

RSA algorithm

Public key

Message ⟶ Encrypted message

Private key

Encrypted message ⟶ Message

RSA algorithm

Message $\xrightarrow{\text{Public key}}$ Encrypted message

Encrypted message $\xrightarrow{\text{Private key}}$ Message

Public

Private

$$n = p \times q$$

large prime numbers

RSA algorithm

Message $\xrightarrow{\text{Public key}}$ Encrypted message

Encrypted message $\xrightarrow{\text{Private key}}$ Message

Public

Private

$n = p \times q$

large prime numbers

Factorising a large number is hard!

Prime factorisation

Prime number - a number that is only divisible by itself and 1, e.g. 2, 3, 5, 7, 11, ...

Every number can be written as a product of its prime factors, e.g.

```
      40
     /  \
   10    4
  /  \  / \
 5   2 2   2
```

40=5x2x2x2

```
   35
  /  \
 7    5
```

35=7x5

To crack RSA, need a factorisation algorithm that finds factors of big numbers quickly

Factorisation algorithm

Algorithm Factor(n):

```
aux <- 2
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux
      else:
        aux <- aux + 1
If aux=n then:
    return "n is prime"
```

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

Want an algorithm that finds a factor p

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2   ⟵ to try as a factor
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux
    else:
        aux <- aux + 1
If aux=n then:
    return "n is prime"
```

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux
    else:
        aux <- aux + 1
If aux=n then:
    return "n is prime"
```

to try as a factor

only try factors less than n

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2                          <- to try as a factor
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux                <- if n is divisible
                                     by aux then we've
                                     found a factor
    else:
        aux <- aux + 1
If aux=n then:
    return "n is prime"
```

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2            ← to try as a factor
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux
    else:
        aux <- aux + 1      ← if n is not divisible by aux then we add 1 to aux and try again
If aux=n then:
    return "n is prime"
```

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2                      to try as a factor
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux
    else:
        aux <- aux + 1
If aux=n then:
    return "n is prime"
```

Use algorithm from earlier - Div(a,b)=yes if a is divisible by b.

if we have tried all the possible factors less than n, then n must be prime

Factorisation algorithm

Algorithm Factor(n):

```
aux <- 2
While aux is less than n do:
     if Div(n,aux)=yes then:
          return aux
        else:
          aux <- aux + 1
If aux=n then:
     return "n is prime"
```

Execute Factor(15):

aux

$\boxed{2}$

$\boxed{3}$

Return 3

Factorisation algorithm

Algorithm Factor(n):

```
aux <- 2
While aux is less than n do:
     if Div(n,aux)=yes then:
          return aux
       else:
          aux <- aux + 1
If aux=n then:
     return "n is prime"
```

Execute Factor(5):

aux

$\boxed{2}$

$\boxed{3}$

$\boxed{4}$

$\boxed{5}$

Return "n is prime"

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2
While aux is less than n do:
     if Div(n,aux)=yes then:
          return aux
        else:
          aux <- aux + 1
If aux=n then:
     return "n is prime"
```

How fast is this algorithm?

$n = p \times q$

p, q prime numbers, $p \leq q$

Algorithm keeps going until aux = p.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2   <-- +1
While aux is less than n do:
      if Div(n,aux)=yes then:
            return aux   <-- +1
         else:
            aux <- aux + 1   <-- +2
If aux=n then:
      return "n is prime"
```

How fast is this algorithm?

$n = p \times q$

p, q prime numbers, $p \leq q$

Algorithm keeps going until $aux = p$.

Factorisation algorithm

```
Algorithm Factor(n):
aux <- 2  ← +1
While aux is less than n do:
    if Div(n,aux)=yes then:
        return aux   ← +1
    else:
        aux <- aux + 1  ← +2
If aux=n then:
    return "n is prime"
```

How fast is this algorithm?

$n = p \times q$

p, q prime numbers, $p \leq q$

Algorithm keeps going until aux = p.

Complexity is at least $2 + (p-2) \times 3$.

Can we make the algorithm faster?

RSA algorithm

When RSA is used, the public key is $n = p \times q$ where p and q are 2048-bit prime numbers.

This means that p and q are about

$2^{2048} = 2^8 \times (2^{10})^{204} \approx$

Useful trick: $2^{10} = 1024 \approx 1000 = 10^3$

RSA algorithm

When RSA is used, the public key is $n = p \times q$ where p and q are 2048-bit prime numbers.

This means that p and q are about

$$2^{2048} = 2^8 \times (2^{10})^{204} \approx 256 \times (10^3)^{204} \approx 10^{614}$$

Useful trick: $2^{10} = 1024 \approx 1000 = 10^3$

RSA algorithm

When RSA is used, the public key is $n = p \times q$ where p and q are 2048-bit prime numbers.

This means that p and q are about

$$2^{2048} = 2^8 \times (2^{10})^{204} \approx 256 \times (10^3)^{204} \approx 10^{614}$$

RSA algorithm

When RSA is used, the public key is $n = p \times q$ where p and q are 2048-bit prime numbers.

This means that p and q are about

$2^{2048} = 2^8 \times (2^{10})^{204} \approx 256 \times (10^3)^{204} \approx 10^{614}$

... which is a VERY big number.

Our factorisation algorithm would take a very very long time to find p and q.

But there is an algorithm which could (in theory) be much faster.

Shor's algorithm

Shor's algorithm is a much faster algorithm for factorising large numbers, but it needs a quantum computer.

Conventional computer:

Bit 0 or 1

Quantum computer:

Qubit 0 or 1 or a 'mixture'

In our factorisation algorithm, the complexity was more than $(p-2) \times 3$.

In Shor's algorithm, the complexity is
(number of digits of $p)^2 \times$ (a constant).

If $p \approx 10^{614}$, this is MUCH faster.

Shor's algorithm

So why is RSA still a safe way to send and receive encrypted messages?

Making quantum computers is hard!

The record for the largest number factorised using Shor's algorithm on a quantum computer is ...

Shor's algorithm

So why is RSA still a safe way to send and receive encrypted messages?

Making quantum computers is hard!

The record for the largest number factorised using Shor's algorithm on a quantum computer is 21.

In 2019, a team tried (and failed) to factorise 35 using Shor's algorithm on an IBM quantum computer.