Prime factorization by Interval-valued computing

Benedek Nagy and Sándor Vályi
University of Debrecen, Hungary

NUMBER THEORY AND ITS APPLICATIONS

OCTOBER 4-8, 2010
DEBRECEN, HUNGARY

Outline

- Motivation
- Interval-valued computations
  - Operations
    - Boolean
    - Shift
    - Product
- Prime factorization
  - Representation of numbers
  - The computation: Subroutines

Motivation

- Intervals
- Computing with intervals
- Nice visualisation, easy to follow

History

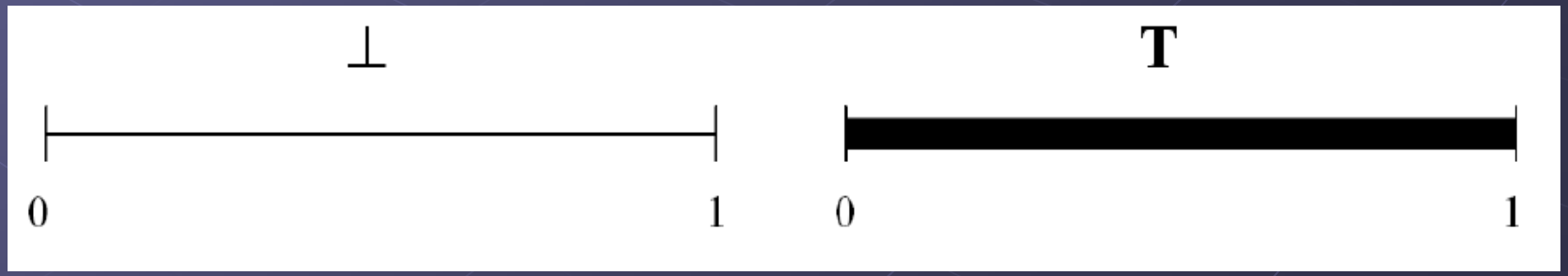
- New computing paradigms: last 20 years:
  - DNA computing
  - Membrane computing
  - Quantum computing
  - Interval-valued computing (2005)
- Cooperation with Sándor Vályi (2006-, PhD dissertation)
  - Defence: Question from Prof. A. Pethő…

Interval-values

Each interval $[a,b)$ with $0 \le a \le b \le 1$ is an atomic interval which contains all the points x between a and b ($a \le x \le b$). Interval-valued byte: bits indexed by [0,1)

Interval-value: a subset of [0,1), finite union of atomic intervals. It can be represented by its characteristic function: $[0,1) \rightarrow \{0,1\}$

Graphical representation:



Classical Computation

- ## Classical Logic

Truth-table of basic logical operators

Name	1st variable	2nd variable	Negation	Conjunction	Disjunction	Implication
Sign	A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$
values	0	0	1	0	0	1
	0	1	1	0	1	1
	1	0	0	0	1	0
	1	1	0	1	1	1

Other Operations are derived:

equivalence: $A \equiv B := (A \rightarrow B) \wedge (B \rightarrow A)$, 'xor': $A \oplus B := A \equiv \neg B$

'nor': $A|B := \neg A \wedge \neg B$, 'nand': $A\&B := \neg A \vee \neg B$.

Classical Computation

- Bits and bytes
  - Bit: unit of Information (answer for a YES/NO question)
  - More bits in a byte $\rightarrow$ more high level the CPU.
  - Logical operators - bitwise:

value of A	negation of A	value of B	conjunction of A and B	disjunction of A and B
$x_1\ x_2\ \ldots\ x_n$	$\neg x_1 \neg x_2 \ldots \neg x_n$	$y_1\ y_2\ \ldots\ y_n$	$x_1\ x_2\ \ldots\ x_n$ $\wedge\ \wedge\ \ldots \wedge$ $y_1\ y_2\ \ldots\ y_n$	$x_1\ x_2\ \ldots\ x_n$ $\vee\ \vee\ \ldots \vee$ $y_1\ y_2\ \ldots\ y_n$

Classical Computation

SHIFT (Shifting bits in a byte)

value of A in binary code	left-shift A	right-shift A
$x_1\ x_2\ \ldots\ x_n$	$x_2\ x_3\ \ldots\ x_n\ 0$	$0\ x_1\ x_2\ \ldots\ x_{n-1}$

Generalising the classical computations

- Turing: the tape alphabet is fixed (in size): the information can be stored in a fix size is limited
- Neumann: the number of bits in a byte is also fixed
- Interval: the information can be stored in a fixed size is not limited (i.e. the density of the memory can be raised unlimitedly)
- Classical logic → fuzzy logic (interval-valued)

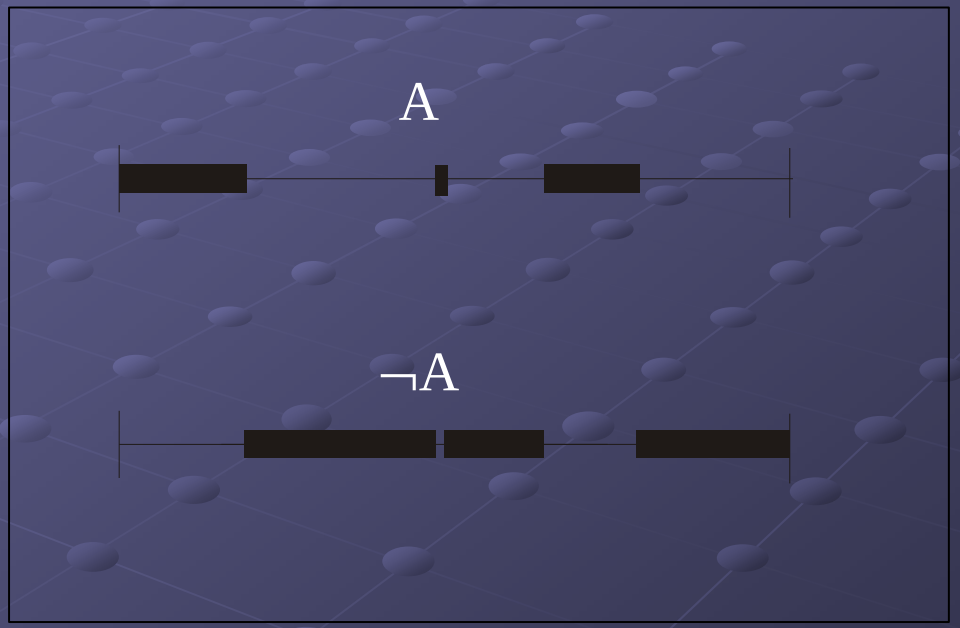
Interval-computation

- Logical gates, circuits:
- Classical computaion is based on classical 2 valued logic.

- Interval-computation is based on interval-valued logic (the number of bits in a unit can be increased during the computation).

BOOLEAN OPERATIONS:
Logic of Interval-values

- Negation

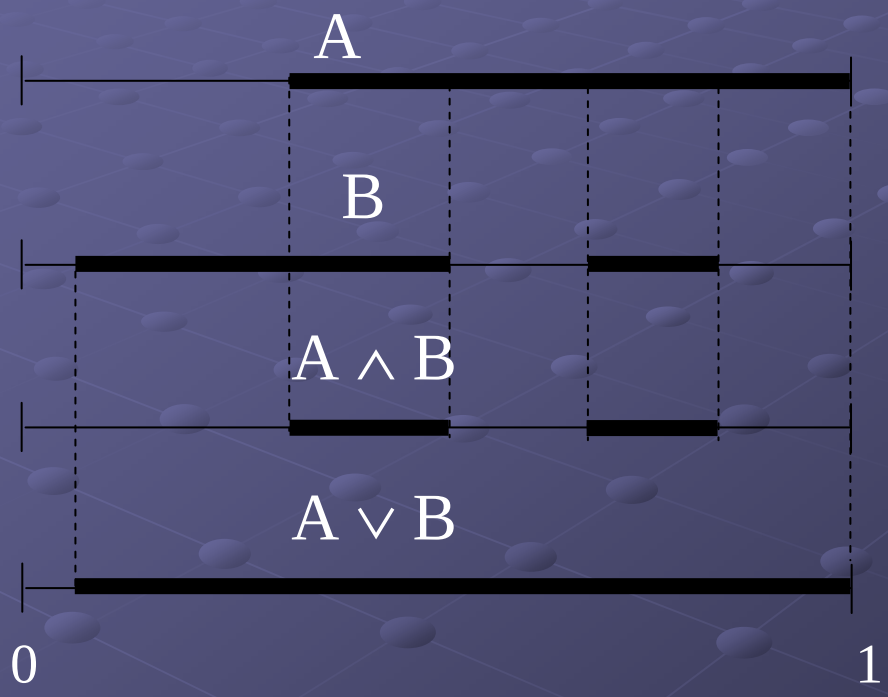


set theory:
complement

- for each point of the Interval we use the negation of the original characteristic value

Logic of Interval-values

- Conjunction and Disjunction



set theory:
intersection
union

for each point of [0,1) we use the logical operation for the characteristic values of the arguments

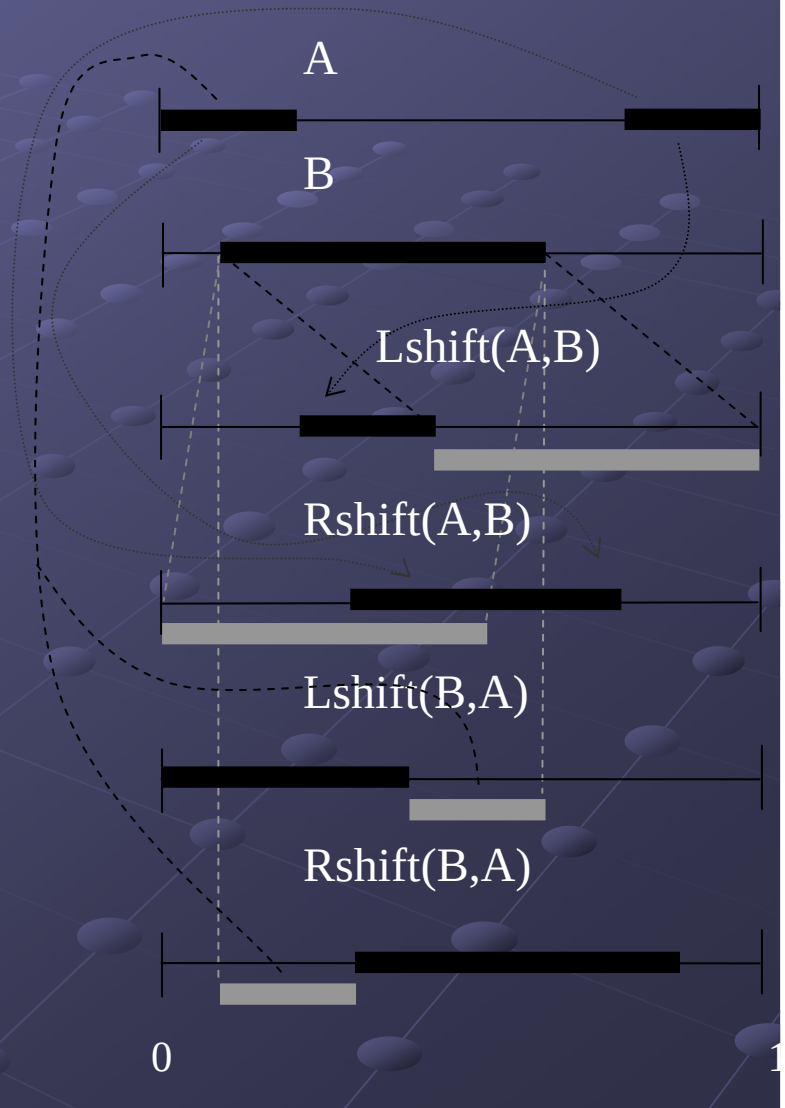
Non-logical operations for Interval-values

- Tool:
  - *Flength*(A) = $b - a$, if A contains the interval $[a,b)$ and A does not contain any interval (a,c) with $c \geq b$, moreover the difference of A and $[a,b)$ does not contain any point x with $x < a$. *Flength*(A) = 0, in other cases.
  - It is the length of the first component of the interval-value A.

Non-logical operations for Interval-values

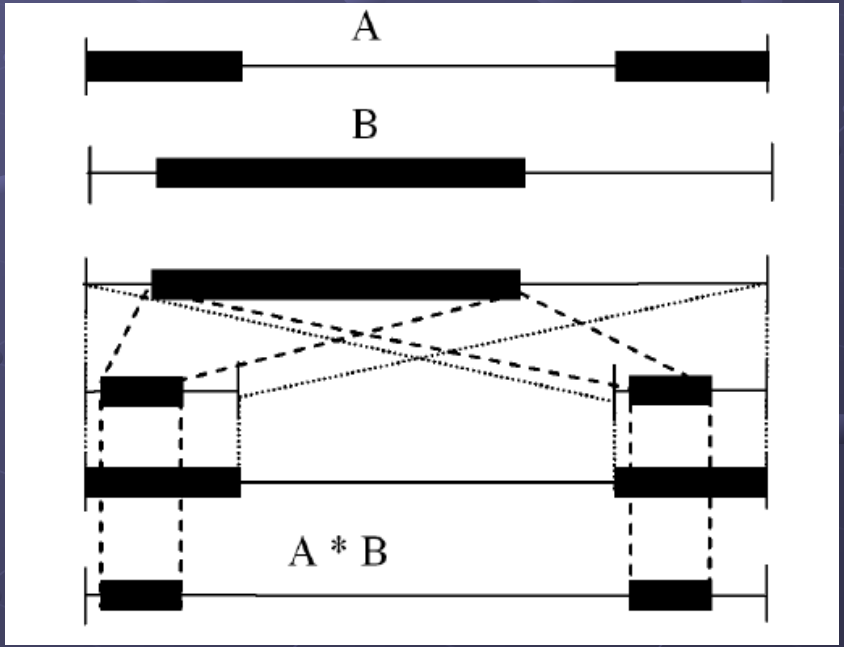
- SHIFT: (we define the 2 directions in a different way to get more effective device)

let Lshift(A,B) and Rshift(A,B) be the interval-values given after A was shifted to the left and to the right by *Flength*(B).



Product of interval-values

- Zooming an interval-value into the complements of the second.



Interval-valued computations

- Computation sequence: sequence of applications of operators to already computed interval-values.
- The sequence starts with the only 1 predefined constant of the system FIRSTHALF : [0,1/2) .
- A language L is *decidable by an interval-valued computation* iff there is an algorithm *A* that for each input word *w* constructs an appropriate computation sequence with last element *A*(*w*) such that ***w*** $\in$ **L if and only if the interval-value of *A*(*w*) is nonempty**.

Computation of functions

- Discrete functions can also be computed by our system.
- Special operation gives 0/1 bit on the output: (*OUTPUT, i)*, where *i is an index less than the index of the actual instruction.*
- $\perp \Rightarrow$ 0, otherwise 1.
- these output bits are concatenated during the computation process.

Visual and effective computation

- Solution to SAT
  - Independent interval-values for the variables
- Can be extended to solution to Q-SAT
  - Dealing with quantifiers

Solving SAT by Interval-values

- Let n be the number of the variables of the SAT.
- Let [0,1/2) be a constant interval-value: FIRSTHALF.

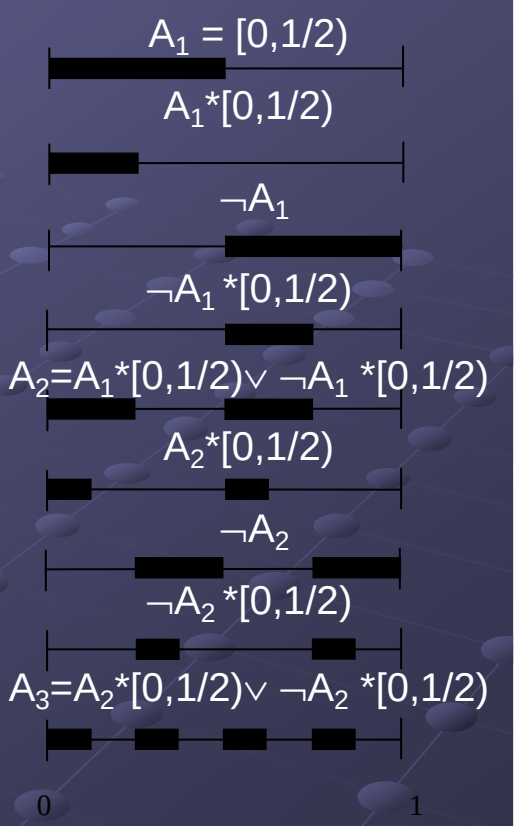
A computing sequence : starting from FIRSTHALF.

- That is: $S_0,\ldots, S_n$, ($n \in \mathbf{N}$),
- where S_0 = FIRSTHALF,
- for each $i<n$: $S_i = \mathrm{op}(S_k, S_j)$, where $k,j<i$.
- Where op can be a Boolean operator, a shift or product. (in negation only the first operand is used)

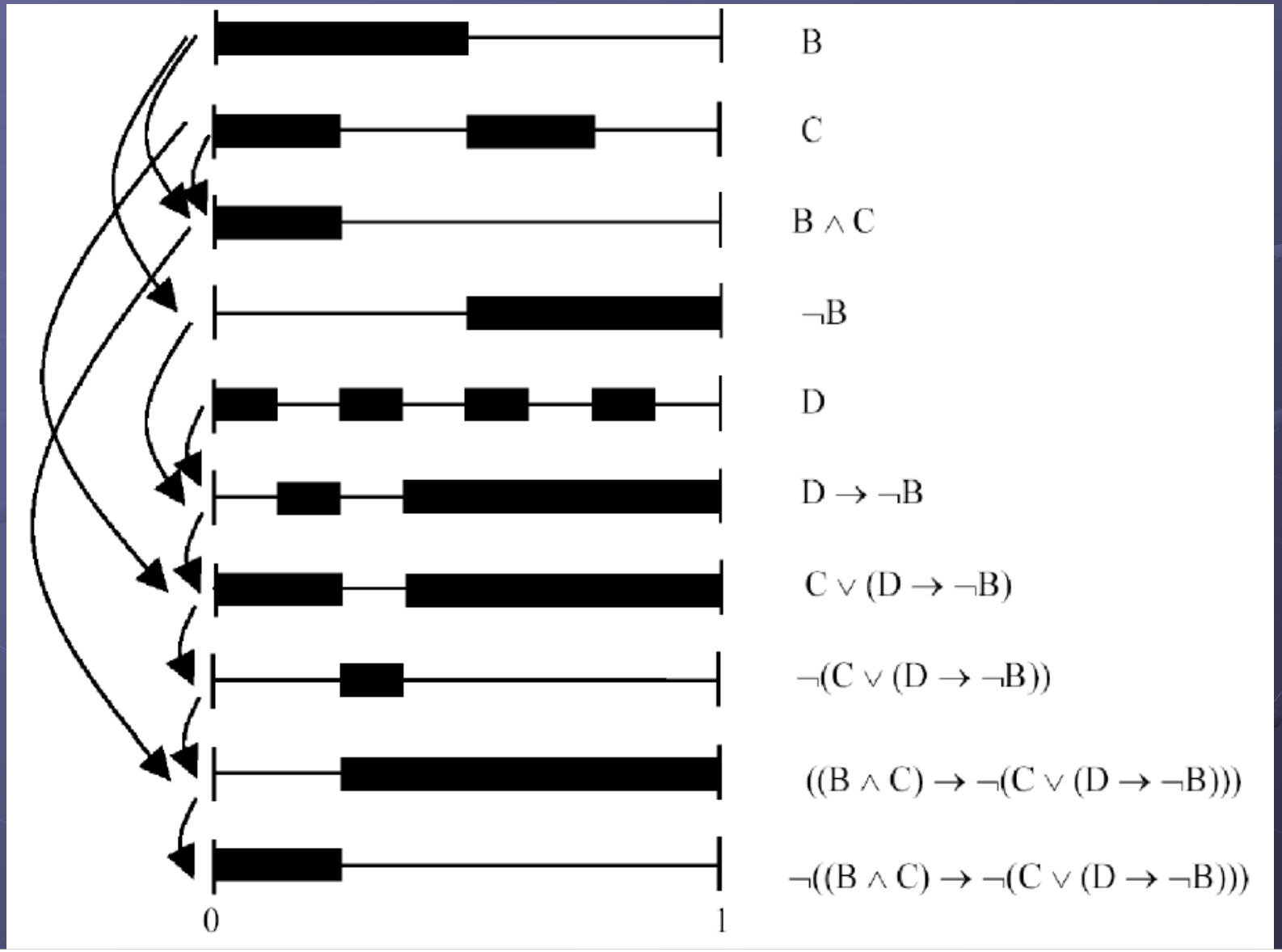
Solving SAT in an effective way

- We need to construct all possible combinations:
- It is possible by product, negation and union. Their number is linear on the number of variables:
- $A_{i+1}=A_i*[0,1/2)\vee\neg A_i*[0,1/2)$
- So, in this way the *i*th variable is:

$$A_i = \bigcup_{j=0}^{2^{i-1}-1} \left[\frac{j}{2^{i-1}}, \frac{2j+1}{2^i}\right)$$



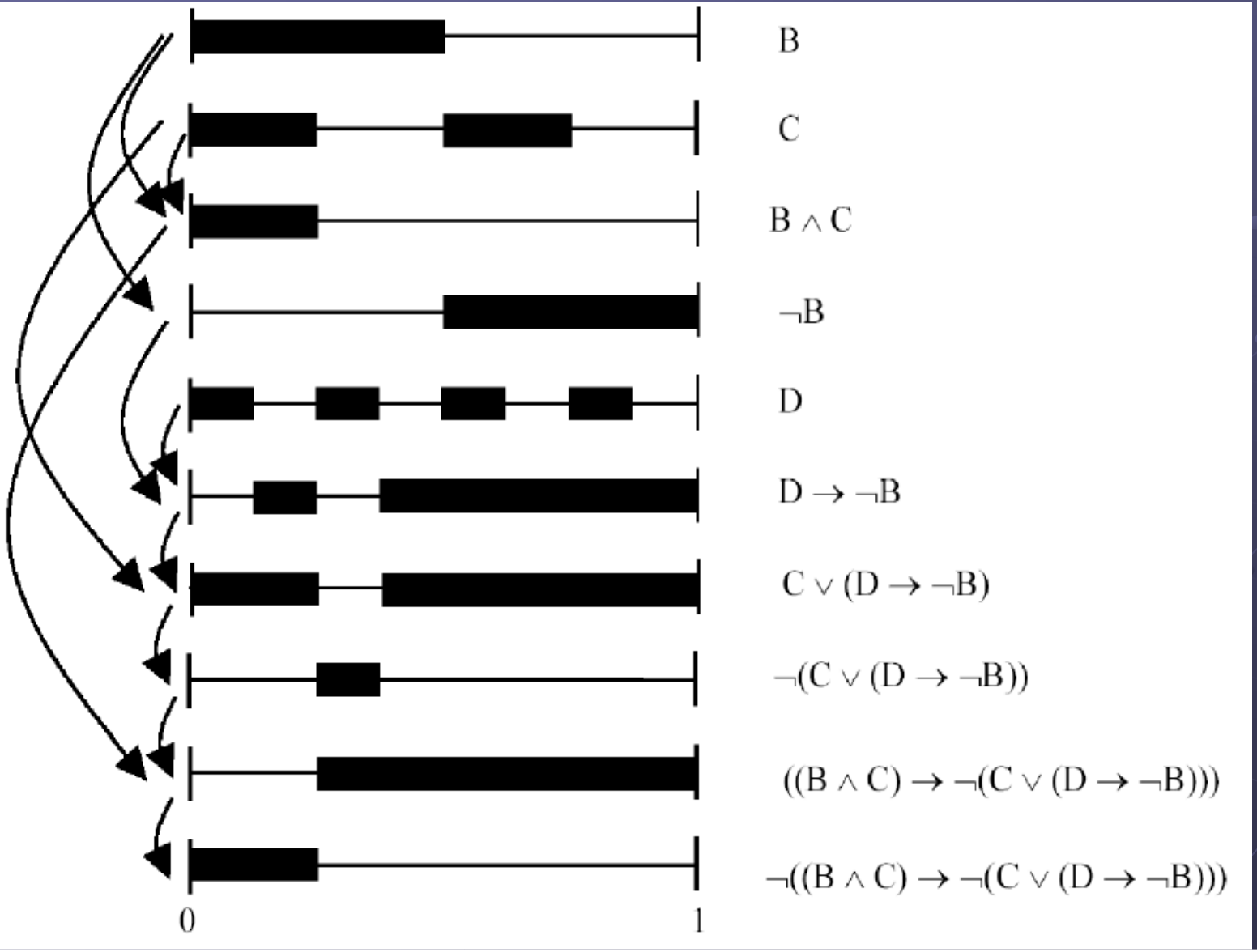
Satisfiability of $\neg((B \wedge C) \rightarrow \neg(C \vee (D \rightarrow \neg B)))$



- Similar to the evaluation by a truth-table, but
- all possible combinations of the truth-values are evaluated parallely.

- Linear solution to SAT.
- The computation is visualised by intervals:
  - Interval-values of independent variables.
  - Logical operations (as in truth table).

Analogous truth-table - knowing the length of the smallest component



11110000
11001100
11000000
00001111
10101010
01011111
11011111
00100000
00111111
11000000

- The PSPACE-complete problem QSAT is solvable by a linear interval-valued computation starting by FIRSTHALF.

Prime factorization

- Every natural number has a unique decomposition to prime numbers.
- Problem of factorization: Having a natural number give a proper divisor, if any…
  - Mathematical interest
  - Applications: Cryptography

  - INPUT: a natural number $n>1$

Representation of numbers

- A number will be represented by a sequence of interval-values, as bites.
- Several numbers can also be represented parallely on these interval-values.
- The input number in binary representation:
  $1 \sim \top$, $0 \sim \bot$
  is used as the input of our algorithm.

Subroutines that we need

- Initialization: $T=[0,1)$ and $\perp$
- Representing the input word
- Creating multiplicands and multipliers
- Multiplying
- Equality test
- Answer: output generation

T and ⊥

- Initial value: FIRSTHALF=[0,1/2)
- [1/2,1) = Rshift(FIRSTHALF, FIRSTHALF)
- T = FIRSTHALF ∨ [1/2,1)
- ⊥ = FIRSTHALF ∧ [1/2,1)

The work of the algorithm is shown by an example

- Input: 6

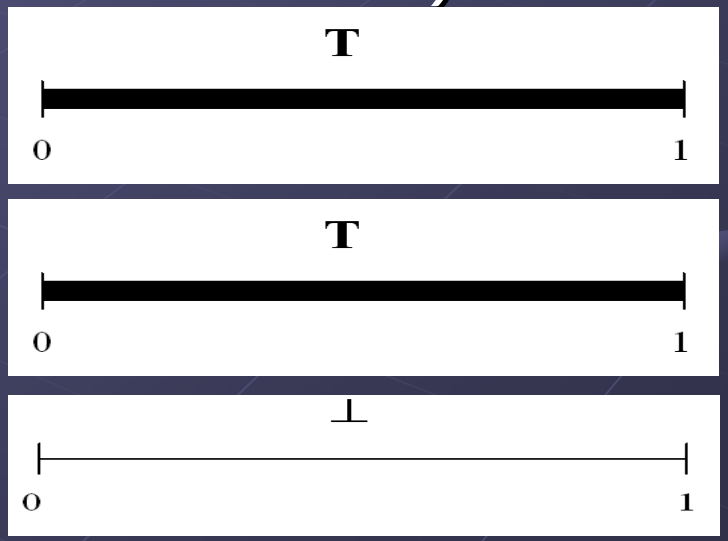
- REPRESENTATION OF THE NUMBER

Binary: 110 (3 bits: 3 interval-values)

A_1

A_2

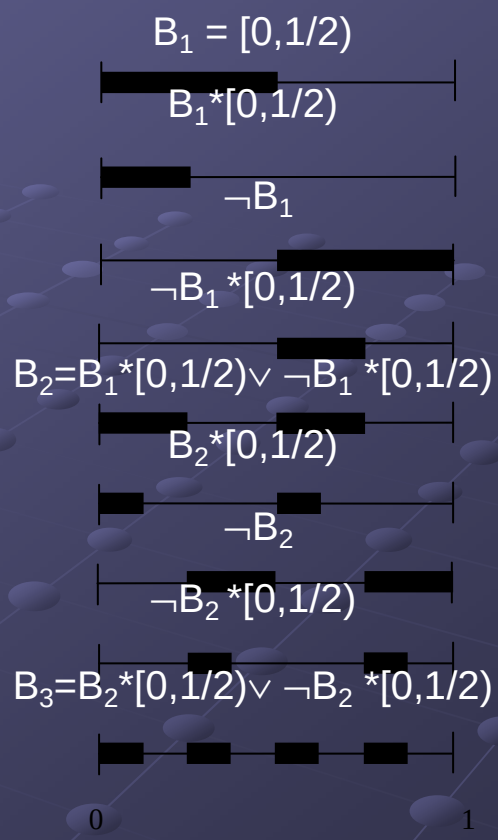
A_3



Construction of possible multipliers

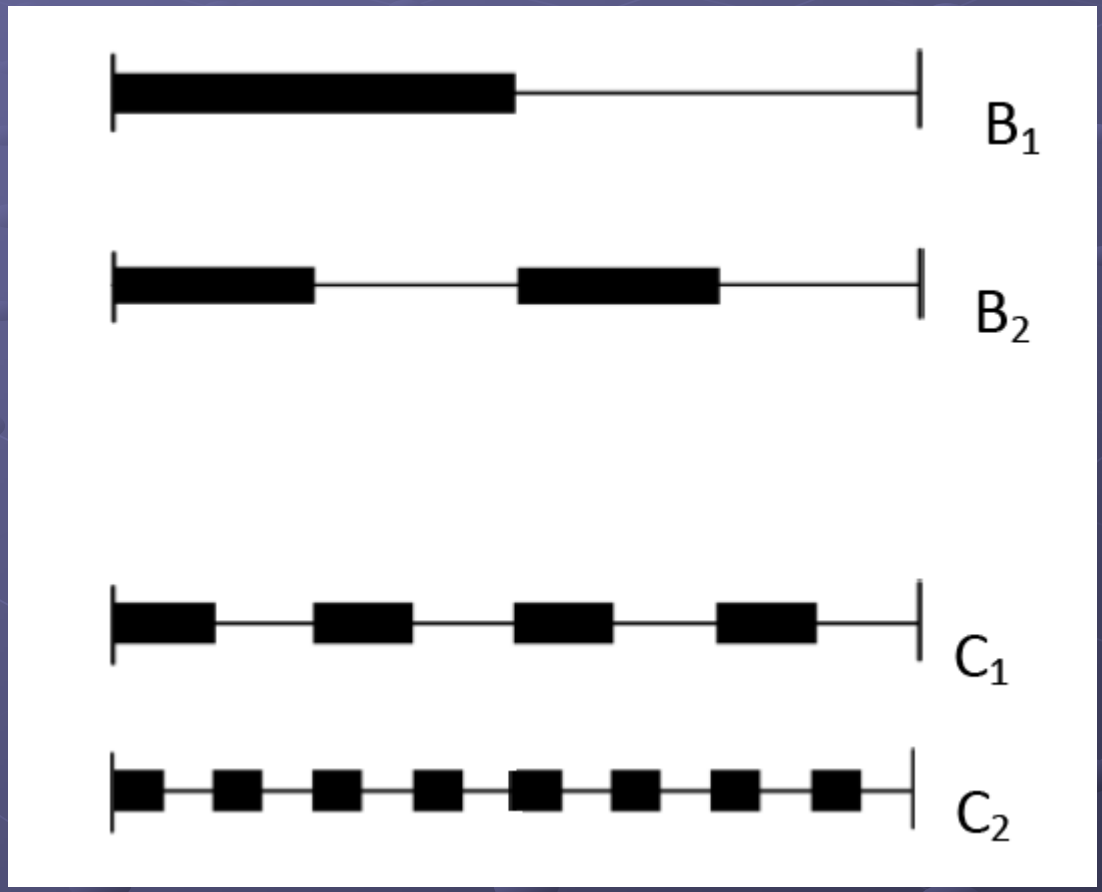
- Same method as in SAT:
- 2 values represented on at most *n-n* bits
- linear on the number of bits:
- $B_{i+1}=B_i*[0,1/2)\vee\neg B_i*[0,1/2)$
- So, in this way the *i*th value is:

$$B_i = \bigcup_{j=0}^{2^{i-1}-1} \left[\frac{j}{2^{i-1}}, \frac{2j+1}{2^i}\right)$$



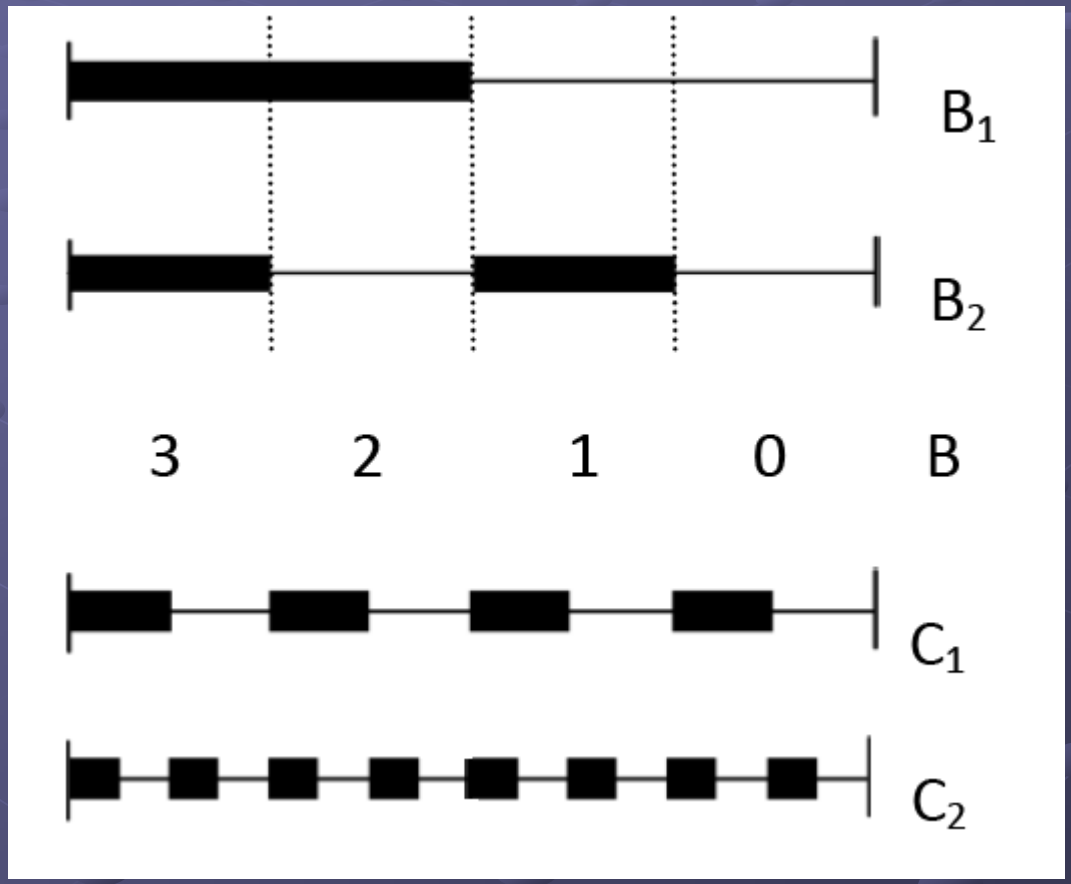
In our example

We will use 2-2 bits:



Various numbers: multipliers and multiplicants

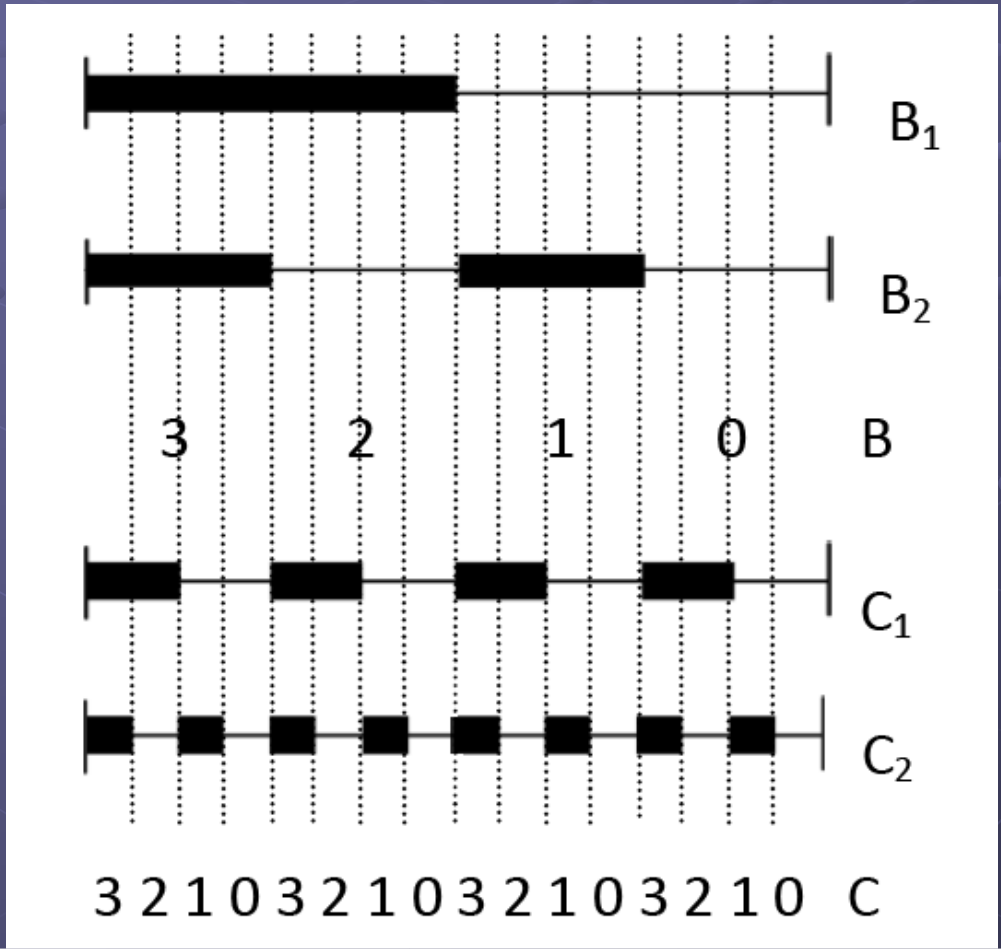
- Vertically: first *n* bits and last *n* bits:



first multiplier

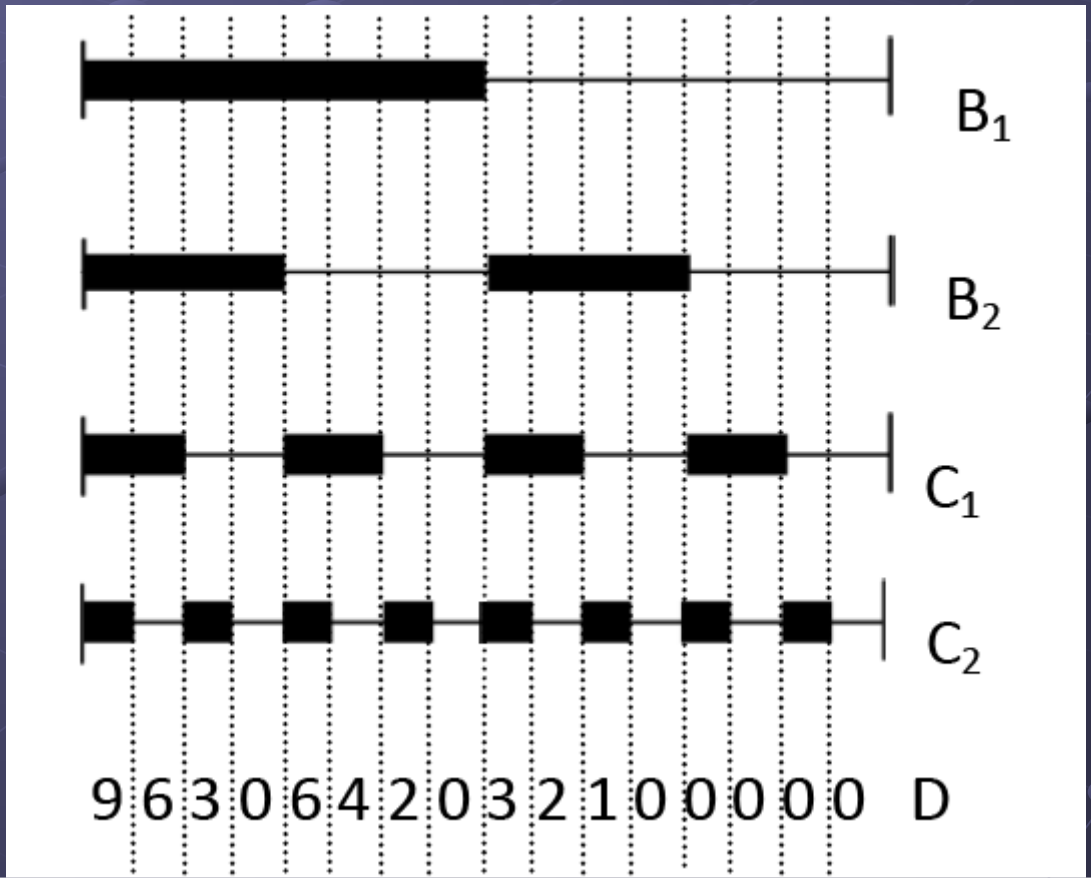
Various numbers: multipliers and multiplicants

- Vertically: first *n* bits and last *n* bits:

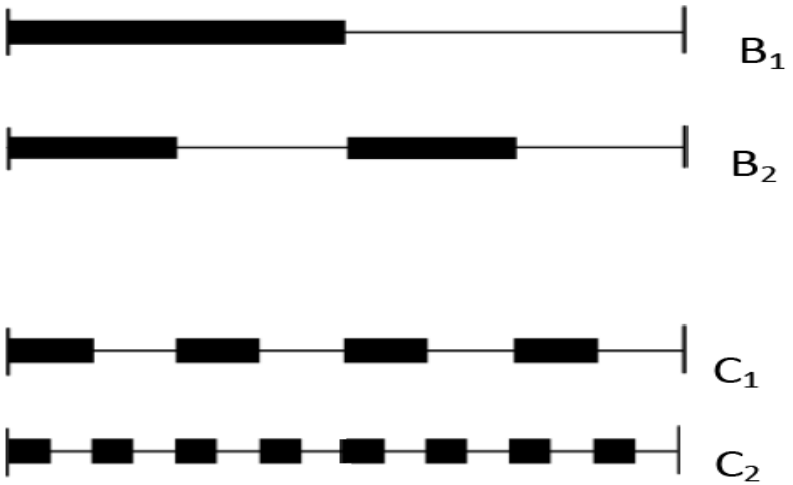


The multiplications

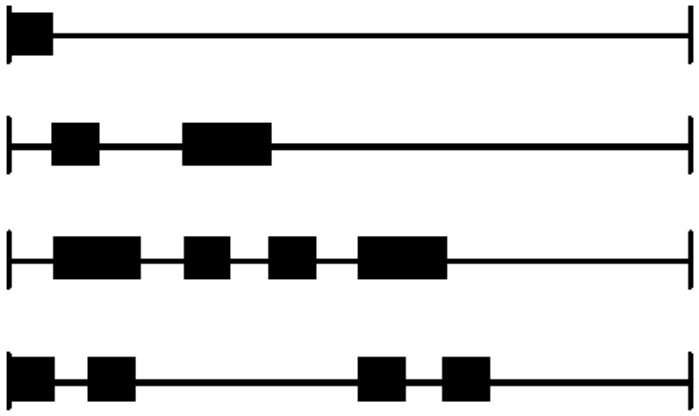
- It is done by logical (bit) operations in ~ square number of operations.
- In our example:



Multiplication by logical operations



9 6 3 0 6 4 2 0 3 2 1 0 0 0 0 D

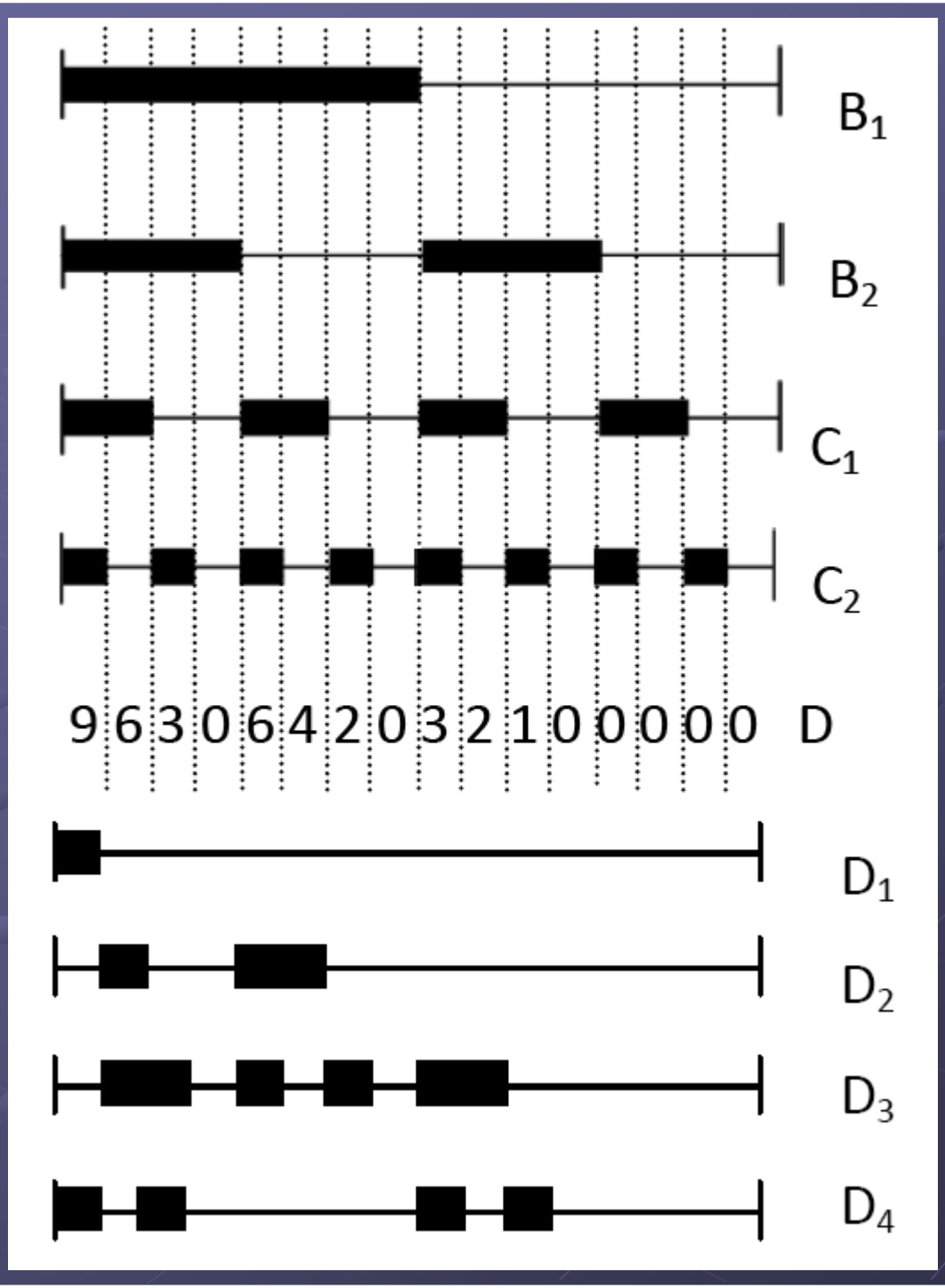


$D_1 = B_1 \wedge C_1 \wedge (B_1 \wedge C_2 \wedge B_2 \wedge C_1) = B_1 \wedge C_2 \wedge B_2 \wedge C_1$

$D_2 = (B_1 \wedge C_1) \oplus (B_1 \wedge C_2 \wedge B_2 \wedge C_1)$

$D_3 = (B_1 \wedge C_2) \oplus (B_2 \wedge C_1)$

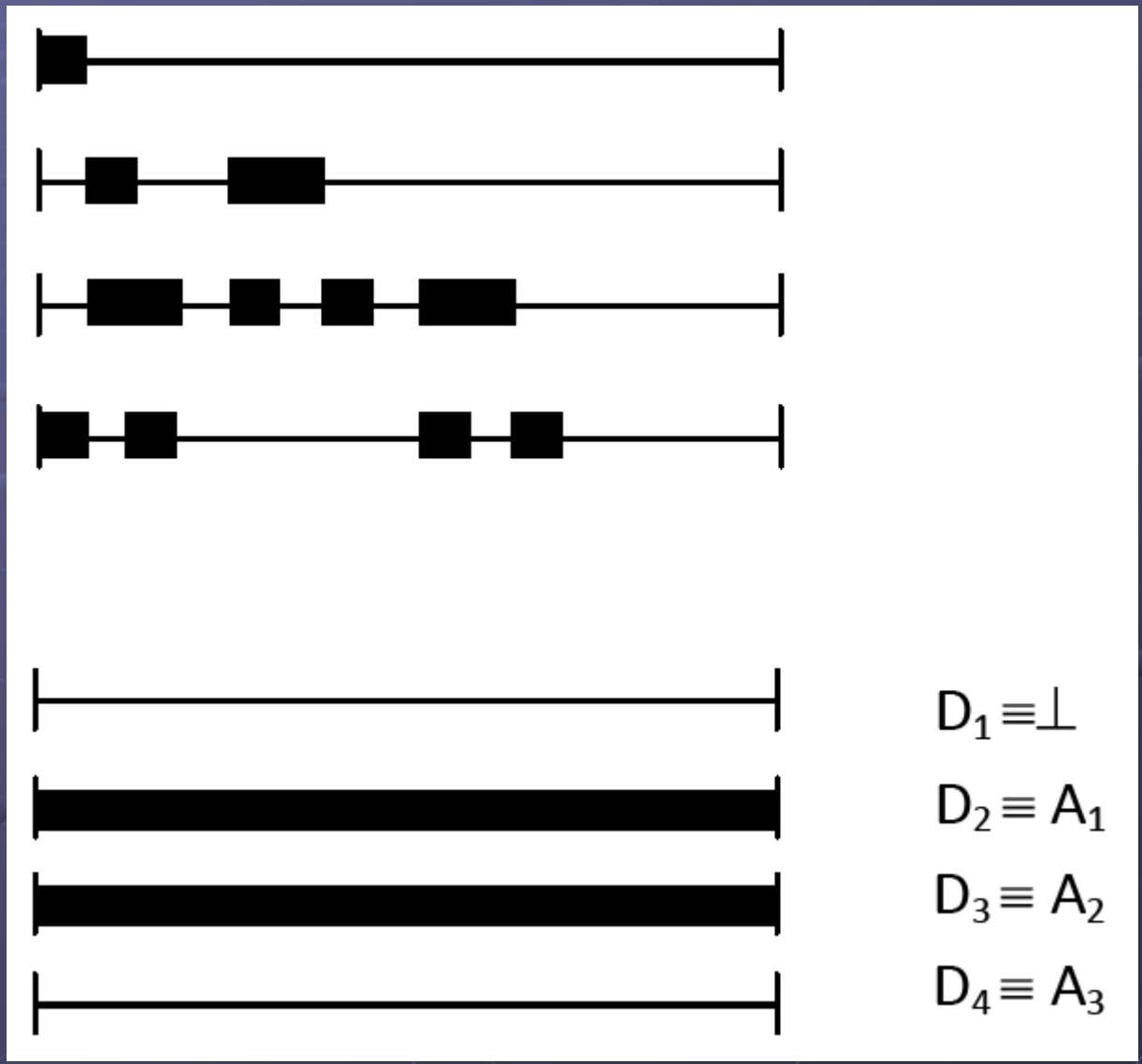
$D_4 = B_2 \wedge C_2$


B1
B2
C1
C2
9 6 3 0 6 4 2 0 3 2 1 0 0 0 0 0 D
D1
D2
D3
D4

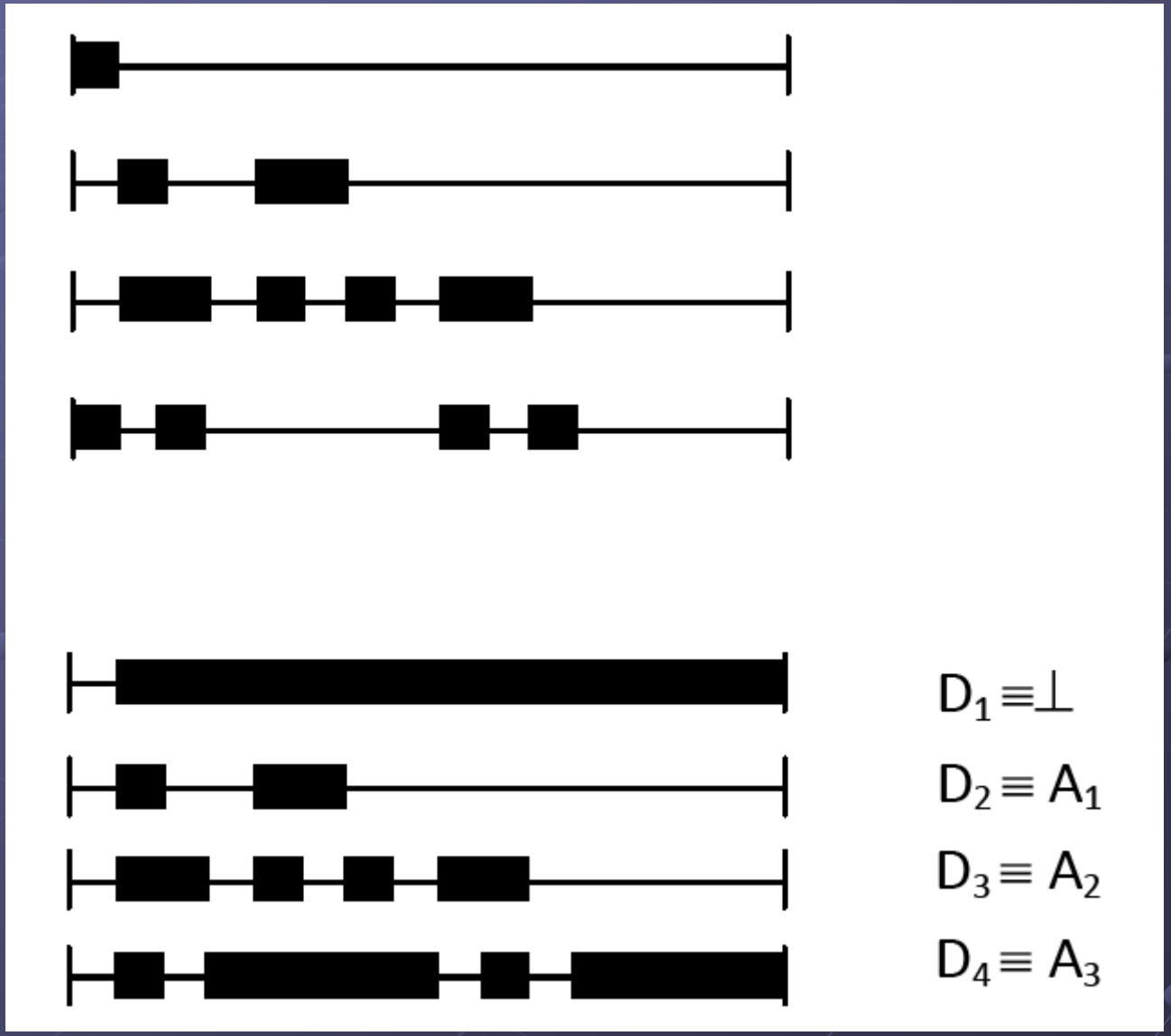
Equality test

- Result of multiplication

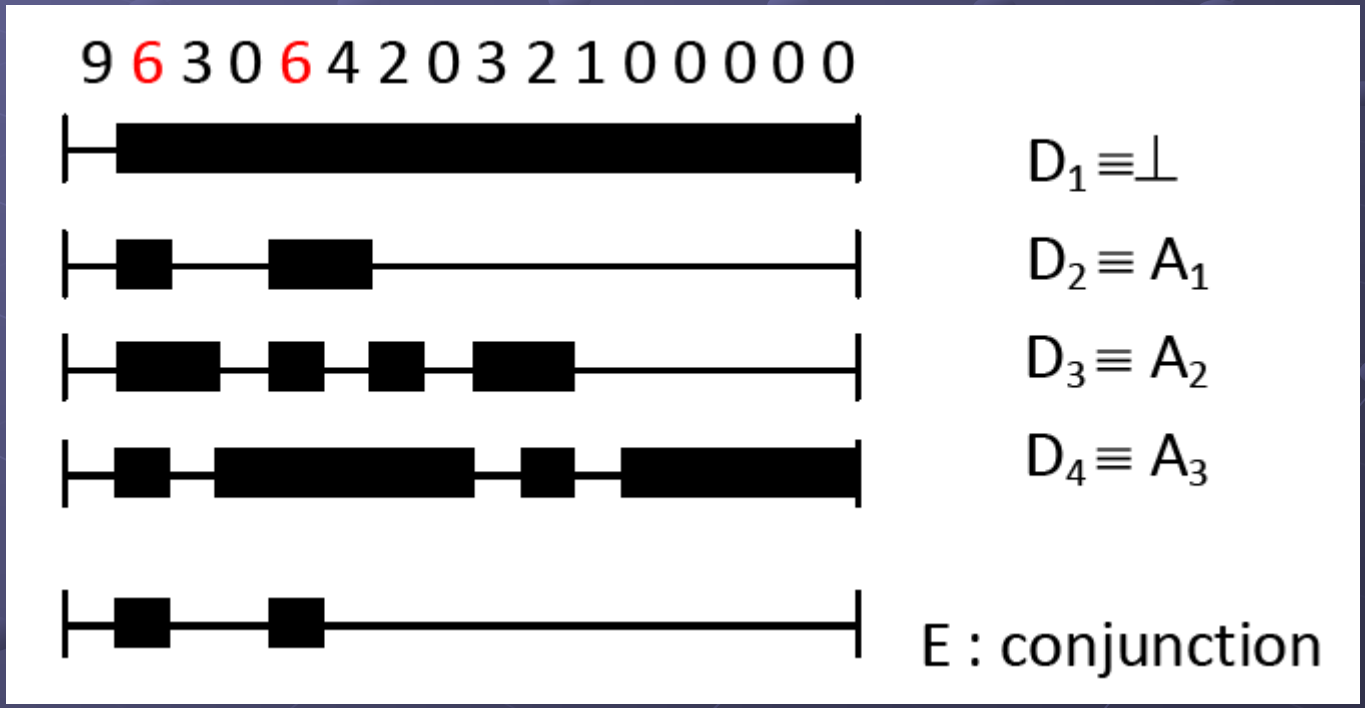
The value we search for



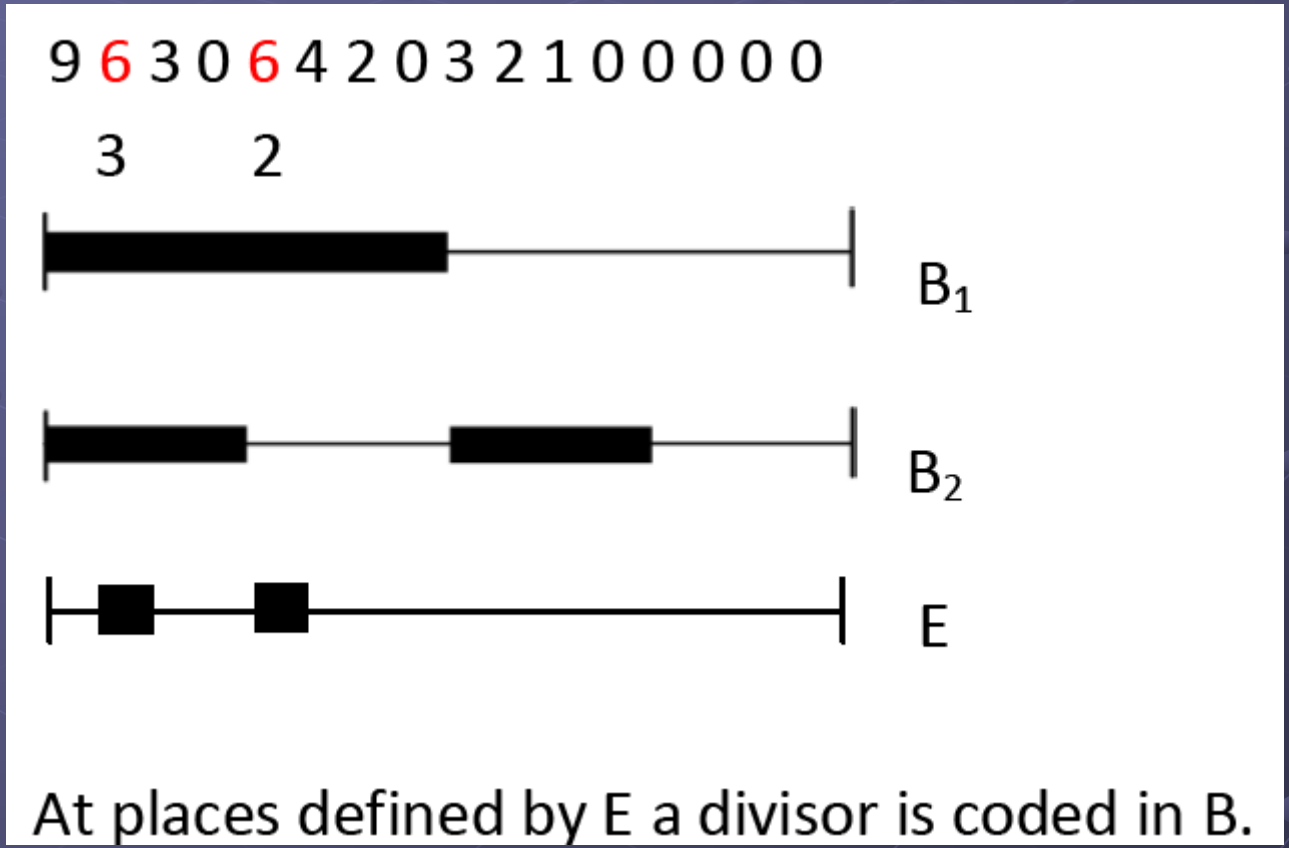
- Result of test



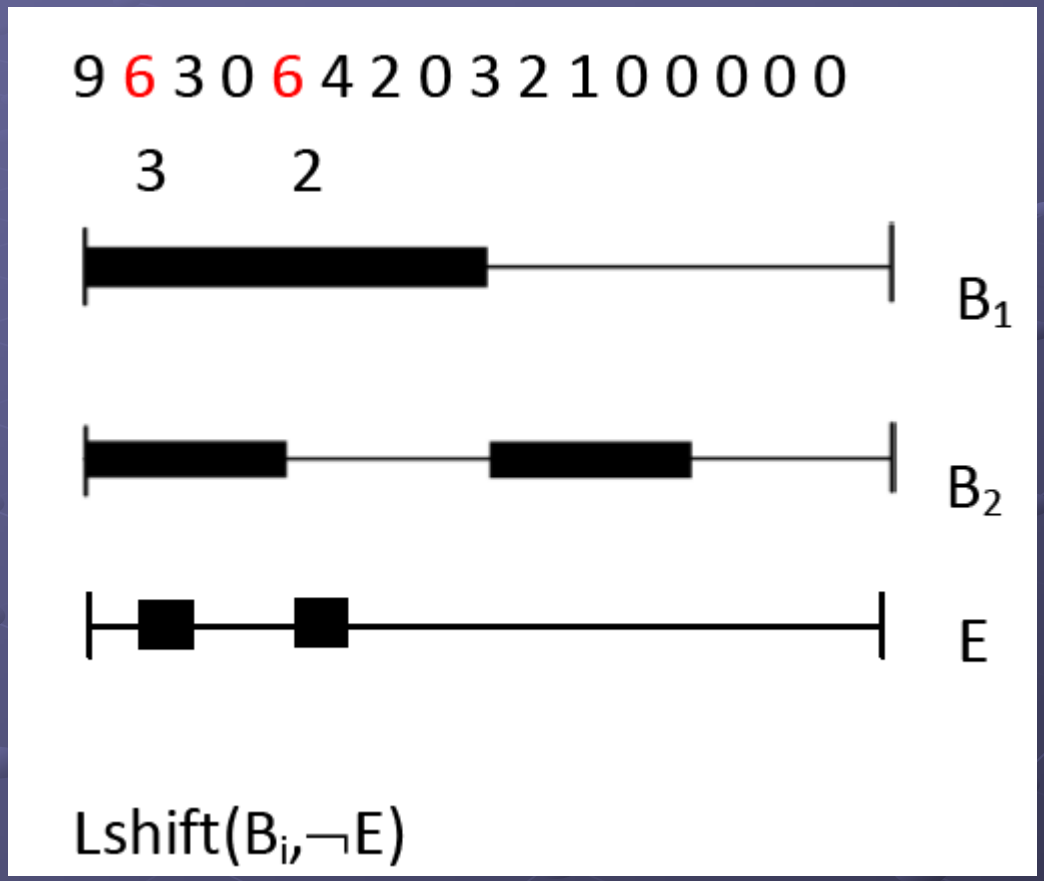
Result of the test: our input can be decomposed



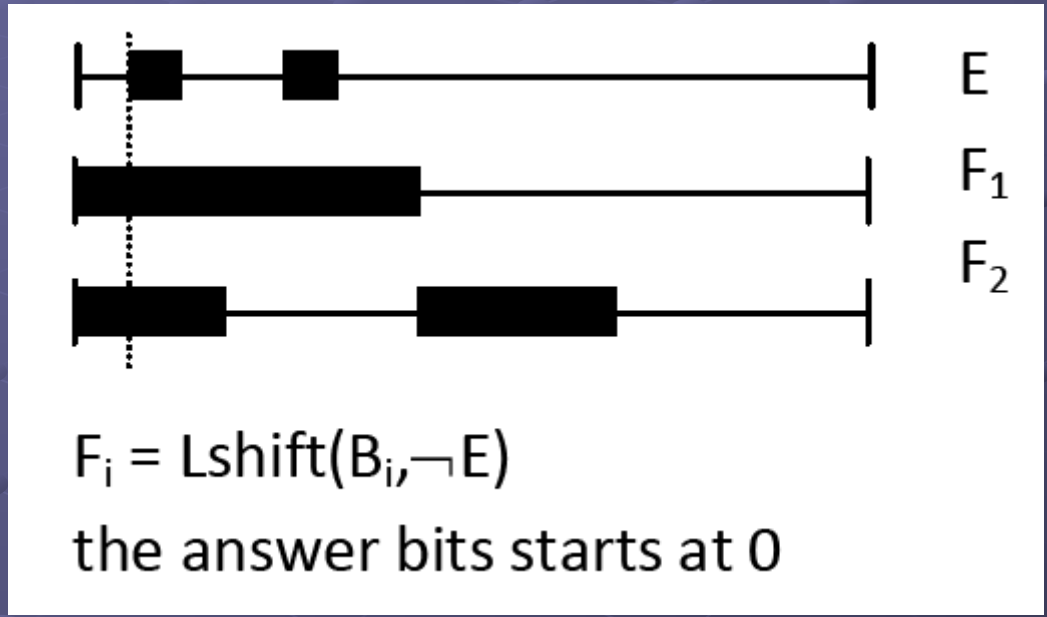
Decoding a divisor



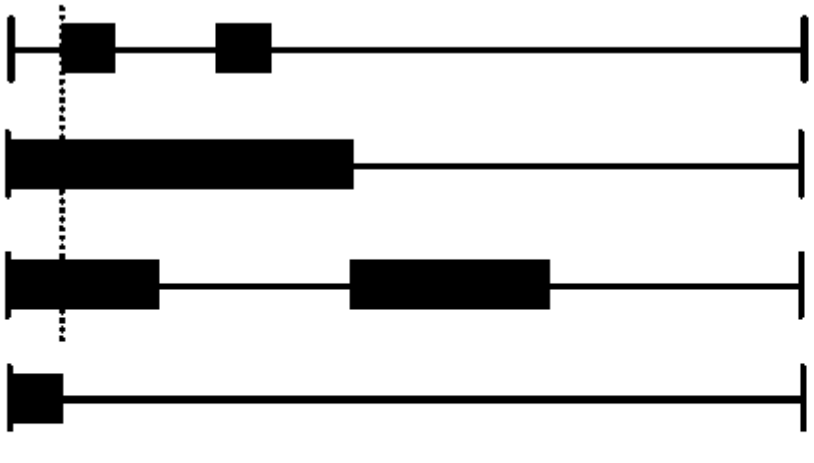
At places defined by E a divisor is coded in B.


9 6 3 0 6 4 2 0 3 2 1 0 0 0 0
3
2
B1
B2
E
Lshift(Bi,¬E)

- The bits of the answer start at 0.



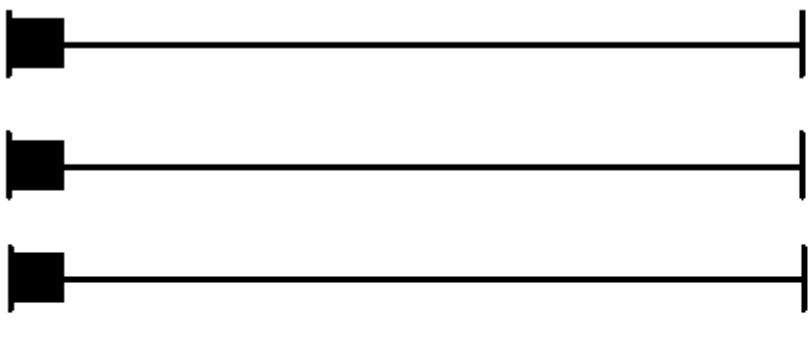
- We need to delete the remaining parts…



$G_0=[0,1/2)*[0,1/2)*\ldots*[0,1/2) = [0,1/2)^{2n}$

$G_i = \text{Lshift}(\text{Rshift}(F_i,\neg G_0),\neg G_0)$

Output the answer



$G_0=[0,1/2)*[0,1/2)*\ldots*[0,1/2) = [0,1/2)^{2n}$

$G_i = \text{Lshift}(\text{Rshift}(F_i,\neg G_0),\neg G_0)$

output is written bitwise:

by G_2 we put 1

by G_1 we put 1

answer: 11, i.e., 3 is a divisor of 6

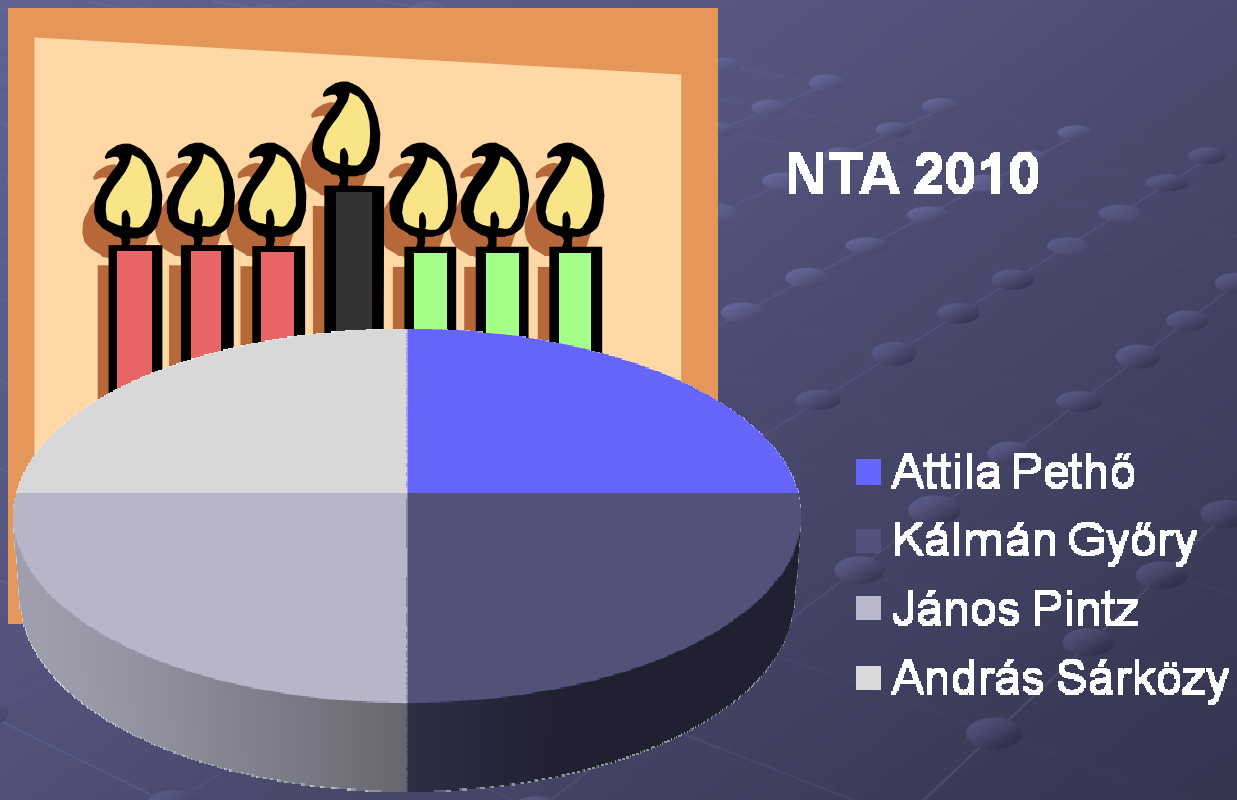
Conclusions

- Intervals can be used to visualize to solution of SAT, and other problems.
- Moreover polynomial number of interval-values is enough to do prime factorization. Effectively it is a uniform solution of the problem: all numbers that can be represented by such number of digits are factorized: only the output sould be changed according to the question.

Thank You for your attention !

AND …

Thank You for your attention !



Happy Birthday to Attila, Kálmán, János and András !