Constant time operations for words of length w

- **Clear i leftmost bits.**

  Algorithm 1:

  > Shift left i bit positions.
  > Shift right i bit positions.

  Example:

$$
\begin{array}{llllllll}
 & a_1 & \cdots & a_i & a_{i+1} & \cdots & a_{w-i} & \cdots & a_w \\
\rightarrow & a_{i+1} & \cdots & a_{2i} & \cdots & a_w & 0 & \cdots & 0 \\
\rightarrow & 0 & \cdots & 0 & a_{i+1} & \cdots & a_{w-i} & \cdots & a_w
\end{array}
$$

  Algorithm 2:

  > AND with the mask 0^i1^{w-i}

  Example:

$$
\begin{array}{l|llllll}
 & a_1 & \cdots & a_i & a_{i+1} & \cdots & a_w \\
\wedge & 0 & \cdots & 0 & 1 & \cdots & 1 \\
\hline
 & 0 & \cdots & 0 & a_{i+1} & \cdots & a_w
\end{array}
$$

- **Concatenate L copies of a string of length k.**
  Assume the string is in the rightmost k positions of a word and $w \geq kL$.

  Algorithm:

  > Multiply by $1(0^{k-1}1)^{L-1}$.

  Example:

$$
\begin{array}{ccccccccc}
 & & & & & & a_1 & a_2 & a_3 \\
 & \times & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\hline
a_1 & a_2 & a_3 & a_1 & a_2 & a_3 & a_1 & a_2 & a_3
\end{array}
$$

- **Expand a string.**
  Given a string a of length k, insert L copies of 0 between each letter.
  Assume $w > (k-1)(L+1)$.

  The following algorithm is for the case $L \geq k$.

  Algorithm 1:

  > Concatenate k copies of the string $0^{L-k}a$
  > Multiply a by $1(0^L1)^{k-1}$
  > AND with the mask $(0^L1)^k$.

  Example:

$$
\begin{array}{cccccccccccc}
 & & & & & & & & & a_1 & a_2 & a_3 \\
\rightarrow & a_1 & a_2 & a_3 & 0 & a_1 & a_2 & a_3 & 0 & a_1 & a_2 & a_3 \\
\wedge & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hline
 & a_1 & 0 & 0 & 0 & 0 & a_2 & 0 & 0 & 0 & 0 & a_3
\end{array}
$$

  The following algorithm is for the case $k/2 \leq L < k$.

Algorithm 2:

> Clear the $k - L$ leftmost bits of a:
>   AND a with the mask $0^{w-L}1^L$
> Insert L copies of 0 between each letter, using Algorithm 1
>
> Shift a right L positions
> Insert L copies of 0 between each letter, using Algorithm 1
> Shift left $L(L+1)$ positions
>
> OR the results of these two subcomputations

Example ($k = 4$, $L = 2$):

$$
\begin{array}{c|cccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & a_1 & a_2 & a_3 & a_4 \\
\to & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_3 & a_4 \\
\to & 0 & 0 & 0 & 0 & 0 & 0 & a_3 & 0 & 0 & a_4
\end{array}
$$

$$
\begin{array}{c|cccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & a_1 & a_2 & a_3 & a_4 \\
\to & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_1 & a_2 \\
\to & 0 & 0 & 0 & 0 & 0 & 0 & a_1 & 0 & 0 & a_2 \\
\to & a_1 & 0 & 0 & a_2 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|cccccccccc}
 & a_1 & 0 & 0 & a_2 & 0 & 0 & 0 & 0 & 0 & 0 \\
\vee & 0 & 0 & 0 & 0 & 0 & 0 & a_3 & 0 & 0 & a_4 \\
\hline
 & a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 & 0 & 0 & a_4
\end{array}
$$

- **Compact a string**
  Given a word a consisting of k fields of length f, with $L \geq (k-1)f$ copies of 0 between each field and $r \geq 0$ copies of 0 to the right of the rightmost field (so $w \geq k(f+L)+r-L$), create a string with the bits of the k fields in the rightmost kf bits of a word.

  Algorithm:

  > Multiply a by $1(0^{L-1}1)^{k-1}$
  > Shift left $w - fk - L(k-1) - r$ positions
  > Shift right $w - fk$ positions

  Example ($f = 2$, $k = 3$, $L = 4$, $r = 0$, $w = 15$):

$$
\begin{array}{c|ccccccccccccccc}
 & & a_6 & a_5 & 0 & 0 & 0 & 0 & a_4 & a_3 & 0 & 0 & 0 & 0 & a_2 & a_1 \\
 & & & & & & \times & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
\hline
 & * & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_4 & a_3 & a_2 & a_1 & 0 & 0 & a_2 & a_1 \\
\to & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_4 & a_3 & a_2 & a_1 & 0 & 0 & a_2 & a_1 & 0 \\
\to & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1
\end{array}
$$

- **Replicate letters within a string.**
  Given a string a with k letters and at least L copies of 0 between each letters, create a new string with $L+1$ adjacent copies of each letter. Assume $k(L+1) \leq w$.

Algorithm:

Multiply a by 1^{L+1}

Example:

	0	0	0	a_1	0	0	0	a_2	0	0	0	a_3
$\times$	0	0	0	0	0	0	0	0	0	1	1	1
	0	a_1	a_1	a_1	0	a_2	a_2	a_2	0	a_3	a_3	a_3

- **Suffix (and Prefix) Sums of Well Spaced Strings**
We say that a string a is *well spaced* if it is of the form $a = 0^L a_1 0^L a_2 \cdots 0^L a_k$, where $k < 2^{L+1}$. The *suffix sum* of a well spaced string is the string $b_k \cdots b_1$ where b_i is the $L+1$ bit binary representation of $a_i + \cdots + a_k$, for $i = 1, \ldots, k$. The condition $k < 2^{L+1}$ is necessary to ensure that each of the sums $a_i + \cdots + a_k$ can be represented using $L+1$ bits.

  The *prefix sum* of a well spaced string is the string $d_1 \cdots d_k$ where d_i is the $L+1$ bit binary representation of $a_1 + \cdots + a_i$, for $i = 1, \ldots, k$. Note that $d_k = b_1$ is the Hamming weight of a.

  The following algorithm computes the prefix and suffix sums of the well spaced string a. It assumes that $w \geq (L+1)(2k-1)$. If $(L+1)k \leq w < (L+1)(2k-1)$, the algorithm can still be used to compute the suffix sums, ignoring any overflow that is generated.

  Algorithm:

  Multiply a by $(0^L 1)^k$

  Example ($k = 3$, $L = 1$):

					0	a_1	0	a_2	0	a_3
				$\times$	0	1	0	1	0	1
					0	a_1	0	a_2	0	a_3
			0	a_1	0	a_2	0	a_3	0	0
$+$	0	a_1	0	a_2	0	a_3	0	0	0	0
	$d_{1,1}$	$d_{1,0}$	$d_{2,1}$	$d_{2,0}$	$b_{1,1}$	$b_{1,0}$	$b_{2,1}$	$b_{2,0}$	$b_{3,1}$	$b_{3,0}$

  Here $b_{i,1}b_{i,0}$ is the binary representation of $a_i + \cdots + a_3$ and $d_{i,1}d_{i,0}$ is the binary representation of $a_1 + \cdots + a_i$.

- **Hamming weight of a well spaced string**, $a = 0^L a_1 0^L a_2 \cdots 0^L a_k$, where $k < 2^{L+1}$ and $w \geq (L+1)k$.

  Algorithm:

  Compute the prefix and suffix sums of the well spaced string
  Shift left $w - (L+1)k$ positions
  Shift right $w - (L+1)$ positions

  Example:

					0	a_1	0	a_2	0	a_3
$\rightarrow$	$c_{1,1}$	$c_{1,0}$	$c_{2,1}$	$c_{2,0}$	$b_{1,1}$	$b_{1,0}$	$b_{2,1}$	$b_{2,0}$	$b_{3,1}$	$b_{3,0}$
$\rightarrow$	$b_{3,1}$	$b_{3,0}$	$b_{2,1}$	$b_{2,0}$	$b_{3,1}$	$b_{3,0}$	0	0	0	0
$\rightarrow$	0	0	0	0	0	0	0	0	$b_{3,1}$	$b_{3,0}$

- **Comparison of two nonnegative integers** $a_1 \cdots a_k$ and $a'_1 \cdots a'_k$ wher $w > k$.

Algorithm 1:

> Subtract $0a'_1 \cdots a'_k$ from $1a_1 \cdots a_k$.
> The leftmost bit is 1 if and only if $a_1 \cdots a_k \geq a'_1 \cdots a'_k$.

Example:

$$\begin{array}{cccccc} & 1 & a_1 & a_2 & a_3 & a_4 \\ - & 0 & a'_1 & a'_2 & a'_3 & a'_4 \\ \hline & b & * & * & * & * \end{array}$$

Here $*$'s denote bits that we don't care about and

$$b = \begin{cases} 1 & \text{if } a_1 \cdots a_4 \geq a'_1 \cdots a'_4 \\ 0 & \text{if } a_1 \cdots a_4 < a'_1 \cdots a'_4 \end{cases}$$

Algorithm 2:

> Subtract $0a'_1 \cdots a'_k$ from $1a_1 \cdots a_k$.
> Subtract 1.
> The leftmost bit is 1 if and only if $a_1 \cdots a_k > a'_1 \cdots a'_k$.

Example:

$$\begin{array}{cccccc} & 1 & a_1 & a_2 & a_3 & a_4 \\ - & 0 & a'_1 & a'_2 & a'_3 & a'_4 \\ \hline & b & c_1 & c_2 & c_3 & c_4 \\ - & 0 & 0 & 0 & 0 & 1 \\ \hline & b' & * & * & * & * \end{array}$$

Here $*$'s denote bits that we don't care about and

$$b' = \begin{cases} 1 & \text{if } a_1 \cdots a_4 > a'_1 \cdots a'_4 \\ 0 & \text{if } a_1 \cdots a_4 \leq a'_1 \cdots a'_4. \end{cases}$$

- **OR of a string** $a = a_1 \cdots a_k$

  Compare a with the number 1, represented by the k-bit string $0^{k-1}1$.

  The following algorithm is for the case $w > k$.

  Algorithm 1:

  > Subtract 0^k1 from $1a_1 \cdots a_k = 2^k + a$.
  > The leftmost bit is 1 if and only if $a_1 \vee \cdots \vee a_k = 1$.

  Example:

  $$\begin{array}{cccccc} & 1 & a_1 & a_2 & a_3 & a_4 \\ - & 0 & 0 & 0 & 0 & 1 \\ \hline & b & * & * & * & * \end{array}$$

  Here $*$'s denote bits that we don't care about and

  $$b = \begin{cases} 1 & \text{if } a_1 \cdots a_4 \geq 1 \\ 0 & \text{if } a_1 = \cdots = a_4 = 0. \end{cases}$$

  The next two algorithms are for the case $w = k$.

Algorithm 2:

> OR a with the mask 10^{k-1}
> Subtract $0^{k-1}1$
> OR with a
> The leftmost bit is 1 if and only if $a_1 \vee \cdots \vee a_k = 1$.

Example:

$$
\begin{array}{c|cccc}
 & a_1 & a_2 & a_3 & a_4 \\
\vee & 1 & 0 & 0 & 0 \\
\hline
 & 1 & a_2 & a_3 & a_4 \\
- & 0 & 0 & 0 & 1 \\
\hline
 & b' & * & * & * \\
\vee & a_1 & a_2 & a_3 & a_4 \\
\hline
 & b & * & * & *
\end{array}
$$

Here $*$'s denote bits that we don't care about and

$$b' = \begin{cases} 1 & \text{if } a_2 a_3 a_4 \geq 1 \\ 0 & \text{if } a_2 = a_3 = a_4 = 0 \end{cases} \quad \text{and } b = b' \vee a_1 = \begin{cases} 1 & \text{if } a_1 \cdots a_4 \geq 1 \\ 0 & \text{if } a_1 = \cdots = a_4 = 0. \end{cases}$$

The following slightly faster algorithm is for the special case when the string a is the binary representation of a number of value at most 2^k, i.e. either $a_1 = 0$ or $a_2 = \cdots = c_k = 0$.

Algorithm 3:

> Subtract a from 10^{k-1}.
> Complement the result. The leftmost bit is 1 if and only if $a_1 \vee \cdots \vee a_k = 1$.

Example:

$$
\begin{array}{c|cccc}
 & 1 & 0 & 0 & 0 \\
- & a_1 & a_2 & a_3 & a_4 \\
\hline
 & \neg b & * & * & * \\
\rightarrow & b & * & * & *
\end{array}
$$

- **Suffix (and Prefix) ORs of Well Spaced Strings**:
  Given a string $a = 0^L a_1 0^L a_2 \cdots 0^L a_k$, where $k < 2^{L+1}$, compute $c = e_1 *^L \cdots *^L e_{k-1} *^L c_1 *^L c_2 *^L \cdots *^L c_k *^L$, where $e_i = a_1 \vee \cdots \vee a_i$ and $c_i = a_i \vee \cdots \vee a_k$, for $i = 1, \ldots, k$. Note that $e_k = c_1$. Here $*$'s denote bits we don't care about. To remove the $*$'s, AND the result with the mask $(10^L)^k$.

  The following algorithm computes the prefix and suffix ORs of the well spaced string a. It assumes that $w \geq (L+1)(2k-1)$. If $(L+1)k \leq w < (L+1)(2k-1)$, the algorithm can still be used to compute the suffix OR's, ignoring any overflow that is generated.

  Algorithm:

  > Compute Suffix Sums
  >   Multiply a by $(0^L 1)^k$
  > Compute k (or $2k-1$) parallel OR's of strings of length $L+1$

Example:

$$
\begin{array}{cccccccccccccccc}
 & & & & & & & & a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 \\
 & & & & & & & \times & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\hline
 & & & & & & & & a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 \\
 & & & & & a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 & 0 & 0 & 0 \\
 & + & a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
 & 0 & d_{11} & d_{10} & 0 & d_{21} & d_{20} & 0 & b_{11} & b_{10} & 0 & b_{21} & b_{20} & 0 & b_{31} & b_{30} \\
\rightarrow & e_1 & * & * & e_2 & * & * & c_1 & * & * & c_2 & * & * & c_3 & * & *
\end{array}
$$

- **Position of leftmost 1 in a string**
  Given a string $a_w \cdots a_1 \neq 0$, return the maximum value of i such that $a_i = 1$.

  Algorithm:

  > Convert from integer to floating point number
  > Extract the exponent field

  This algorithm gives the position of the leftmost one, with positions numbered from the right end of the string. To obtain the position numbered from the left end of the string, subtract the result from $w + 1$.

  Example:

$$
\begin{array}{cccccccccl}
 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \\
\rightarrow & 1 & 1 & 0 & 0 & 1 & 0 & 1 & = .1100 \times 2^5 \\
\wedge & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & = 5
\end{array}
$$

  Here we assume that a floating point number is represented with 4 bits for the fraction followed by 3 bits for the exponent.

- **Position of rightmost 1 in a string**
  Given a string $a_w \cdots a_1 \neq 0$, return the minimum value of i such that $a_i = 1$.

  The following algorithm assumes that $w = uv$, where $u \leq v \leq 2^{u-1}$. For example, if $w = 16$, then $u = v = 4 \leq 2^{4-1}$ and, if $w = 32$, then $u = 4$ and $v = 8 \leq 2^{4-1}$.

  The idea is to view words of w bits as consisting of v blocks of u bits. First, the rightmost block containing a 1 is identified. Then the location of the rightmost 1 in that block is identified.

  A similar algorithm can be used to find the position of the leftmost 1 in a string.

Algorithm:

1 Compute v parallel ORs of strings of length u to get $c = c_v 0^{u-1} \cdots c_1 0^{u-1}$, with $c_i = a_{iu} \vee \cdots \vee a_{(i-1)u+1}$:
  OR a with the mask $(10^{u-1})^v$.
  Subtract $(0^{u-1}1)^v$.
  OR with a.
  AND with $(10^{u-1})^v$.

2 Compute the suffix sums of c:
  Shift c right $u-1$ positions.
  Multiply by $(0^{u-1}1)^v$ to get $b = b_v \cdots b_1$, where b_i is the u-bit binary representation of $c_1 + \cdots + c_i \leq i \leq v \leq 2^{u-1}$.

3 Compute the suffix ORs of c to get $d_v 0^{u-1} \cdots d_1 0^{u-1}$, where $d_i = c_i \vee \cdots \vee c_1$:
  Subtract b from $(10^{u-1})^v$.
  Complement the result.

4 Compute the Hamming weight h of d.
  Shift d right $u-1$ positions.
  Multiply by $(0^{u-1}1)^v$ to get $e = e_v \cdots e_1$, where e_i is the u-bit binary representation of $d_1 + \cdots + d_i$.
  Shift e right $w-u$ positions to get $h = e_v$.

If the kth block is the rightmost block containing 1, (i.e. $c_k = 1$ and $c_1 = \cdots = c_{k-1} = 0$), then $d_1 = \cdots = d_{k-1} = 0$ and $d_k = \cdots = d_v = 1$. Hence, $h = e_v = d_1 + \cdots + d_v = v + 1 - k$.

5 Select the kth block, i.e. $a' = 0^{w-u} a'_u \cdots a'_1$, where $a'_i = a_{(k-1)u+i}$ for $i = 1, \ldots, u$.
  Shift a left $(h-1)u$ positions.
  Shift right $w-u$ positions.

6 Expand a' by inserting $v-1$ copies of 0 between each letter.
  When $u < v$, use algorithm 1.
  When $u = v$, the more complicated algorithm 2 must be used, since $v - 1 < u$.

7 Compute d', the suffix ORs of a'.

8 Compute the Hamming weight h' of d'.

9 Compute $(v + 2 - h)u + 1 - h'$.

Example ($u = 3$, $v = 4 \leq 2^{3-1}$):

$$
\begin{array}{cccccccccccccl}
 & a_{12} & a_{11} & a_{10} & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
\rightarrow & c_4 & 0 & 0 & c_3 & 0 & 0 & c_2 & 0 & 0 & c_1 & 0 & 0 & \\
\rightarrow & 0 & 0 & c_4 & 0 & 0 & c_3 & 0 & 0 & c_2 & 0 & 0 & c_1 & \\
\times & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \\
\cline{2-13}
 & b_{42} & b_{41} & b_{40} & b_{32} & b_{31} & b_{30} & b_{22} & b_{21} & b_{20} & b_{12} & b_{11} & b_{10} & \\
\rightarrow & d_4 & 0 & 0 & d_3 & 0 & 0 & d_2 & 0 & 0 & d_1 & 0 & 0 & \\
\rightarrow & 0 & 0 & d_4 & 0 & 0 & d_3 & 0 & 0 & d_2 & 0 & 0 & d_1 & \\
\times & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \\
\cline{2-13}
 & e_{42} & e_{41} & e_{40} & e_{32} & e_{31} & e_{30} & e_{22} & e_{21} & e_{20} & e_{12} & e_{11} & e_{10} & \\
\rightarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & e_{42} & e_{41} & e_{40} & = h
\end{array}
$$

$$
\begin{array}{cccccccccccccl}
 & a_{12} & a_{11} & a_{10} & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
\rightarrow & a'_3 & a'_2 & a'_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\rightarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a'_3 & a'_2 & a'_1 & \\
\rightarrow & 0 & 0 & 0 & a'_3 & 0 & 0 & 0 & a'_2 & 0 & 0 & 0 & a'_1 & \\
\times & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \\
\cline{2-13}
 & b'_{42} & b'_{41} & b'_{40} & b'_{32} & b'_{31} & b'_{30} & b'_{22} & b'_{21} & b'_{20} & b'_{12} & b'_{11} & b'_{10} & \\
\rightarrow & d'_3 & 0 & 0 & 0 & d'_2 & 0 & 0 & 0 & d'_1 & 0 & 0 & 0 & \\
\rightarrow & 0 & 0 & 0 & d'_3 & 0 & 0 & 0 & d'_2 & 0 & 0 & 0 & d'_1 & \\
\times & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \\
\cline{2-13}
 & e'_{33} & e'_{32} & e'_{31} & e'_{30} & e'_{23} & e'_{22} & e'_{21} & e'_{20} & e'_{13} & e'_{12} & e'_{11} & e'_{10} & \\
\rightarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & e'_{33} & e'_{32} & e'_{31} & e'_{30} & = h'
\end{array}
$$

- **Increment (decrement) a field**
  Given a string a containing fields of length L, increment the k'th least significant field.

  Algorithm:

  > Shift 1 left $L(k-1)$ positions.
  > Add to (subtract from) a.

- **Move the unique non-zero field in each group to the leftmost field in that group**
  Given a string $a = a_{1,1} \cdots a_{1,k} \cdots a_{g,1} \cdots a_{g,k}$ consisting of g groups of k fields, each of length L, with at most one non-zero field in each group (i.e. for each i, there exists m_i such that $a_{i,j} = 0$ for $j \neq m_i$), compute a string with the non-zero field in each group as far left as possible (i.e. $a_{1,m_1} 0^{(k-1)L} \cdots a_{g,m_g} 0^{(k-1)L}$). Assume that all the fields contain a number less than 2^{L-1} (i.e. with most significant bit 0).

  Note that the j'th field of the product is the sum of the j'th field of a and the $k-1$ fields to its right. In particular, the most significant field of the i'th group is equal to $a_{i,1} + \cdots + a_{i,k} = a_{i,m_i}$. The j'th most significant field of the i'th group is equal to $a_{i,j} + \cdots + a_{i,k} + a_{i-1,1} + \cdots + a_{i-1,j-1} \leq a_{i,m_i} + a_{i-1,m_{i-1}} < 2^L$. Thus one field doesn't interfere with an adjacent field during the addition of the partial products in the multiplication.

  > Multiply by $(0^{L-1}1)^k$
  > AND with the mask $(1^L 0^{(k-1)L})^g$

  Example ($g = 2$, $k = 3$, $L = 2$):

	0 $a_{1,1}$	0 $a_{1,2}$	0 $a_{1,3}$	0 $a_{2,1}$	0 $a_{2,2}$	0 $a_{2,3}$
			$\times$	0 1	0 1	0 1
	$a_{1,1}+a_{1,2}+a_{1,3}$	$a_{1,2}+a_{1,3}+a_{2,1}$	$a_{1,3}+a_{2,1}+a_{2,2}$	$a_{2,1}+a_{2,2}+a_{2,3}$	$a_{2,2}+a_{2,3}$	$a_{2,3}$
$\wedge$	1 1	0 0	0 0	1 1	0 0	0 0
	0 a_{1,m_1}	0 0	0 0	0 a_{2,m_2}	0 0	0 0

- **Index of a field beginning with 1**

  Given a string $a = a_1 \cdots a_k$ consisting of k fields of length L, return an index i such that $a_i \geq 2^{L-1}$.

  The following algorithm finds the leftmost field beginning with 1. It is most useful when $L = 2^l$ is a power of 2, in which case the division can be replaced by a right shift of l positions.

  Algorithm 1:

  > AND a with the mask $(10^{L-1})^k$.
  > Find the position of the leftmost 1 in the string, counting from the right end.
  > Divide by L.

  The following algorithm also finds the leftmost field beginning with 1.

  Algorithm 2:

  > AND a with the mask $(10^{L-1})^k$.
  > Compact the bits that indicate which elements won all their comparisons:
  > Multiply by $1(0^{L-2}1)^{k-1}$.
  > Shift left $w - kL$ positions.
  > Shift right $w - k$ positions.
  > Find the position of the leftmost 1 in the string, counting from the right end.

  Example ($k = 3$, $L = 4$):

	*	*	a_1	*	*	*	a_2	*	*	*	a_3	*	*	*
$\wedge$	0	0	1	0	0	0	1	0	0	0	1	0	0	0
	0	0	a_1	0	0	0	a_2	0	0	0	a_3	0	0	0
							$\times$	1	0	0	1	0	0	1
	*	*	a_1	a_2	a_3	0	a_2	a_3	0	0	a_3	0	0	0
$\rightarrow$	a_1	a_2	a_3	0	a_2	a_3	0	0	a_3	0	0	0	0	0
$\rightarrow$	0	0	0	0	0	0	0	0	0	0	0	a_1	a_2	a_3

  If there is no instruction for converting integers to floating point numbers, the following algorithm is more efficient. It finds the rightmost field beginning with 1 using a technique similar to the first 4 steps of the algorithm for computing the position of the rightmost 1 in a string. It assumes that $k < 2^L$ (so that the string is well spaced).

  Algorithm 3:

  > AND a with the mask $(10^{L-1})^k$.
  > Shift right $L - 1$ positions.
  > Compute the suffix ORs of the well spaced string.
  > Shift right $L - 1$ positions.
  > Compute the Hamming weight of the well spaced string.

Example ($L = 3$, $k = 4 \leq 2^{3-1}$):

	a_1	$*$	$*$	a_2	$*$	$*$	a_3	$*$	$*$	a_4	$*$	$*$
$\wedge$	1	0	0	1	0	0	1	0	0	1	0	0
	a_1	0	0	a_2	0	0	a_3	0	0	a_4	0	0
$\rightarrow$	0	0	a_1	0	0	a_2	0	0	a_3	0	0	a_4
$\rightarrow$	d_1	0	0	d_2	0	0	d_3	0	0	d_4	0	0
$\rightarrow$	0	0	d_1	0	0	d_2	0	0	d_3	0	0	d_4
$\rightarrow$	0	0	0	0	0	0	0	0	0	h_2	h_1	h_0

Here $d_i = a_i \vee \cdots \vee a_k$, for $i = 1, \ldots, k$, and $h_2h_1h_0$ is the binary representation of $d_1 + d_2 + d_3 + d_4$. For example, if $a_1 = 1$ and $a_2 = a_3 = a_4 = 0$, then $d_1 = 1$ and $d_2 = d_3 = d_4 = 0$, so $d_1 + d_2 + d_3 + d_4 = 1$. If $a_3 = 1$ and $a_4 = 0$, then $d_1 = d_2 = d_3 = 1$ and $d_4 = 0$, so $d_1 + d_2 + d_3 + d_4 = 3$.

- **Maximum field in a string**

  Given a string $a = a_1a_2 \cdots a_k$, where $a_i = 0c_{i,L-1} \cdots c_{i,1}$ is a string of length L with most significant bit 0, for $i = 1, \ldots, k$, return the location of a maximum field, i.e. return an index $m \in \{1, \ldots, k\}$ such that $a_m = \max\{a_i \mid i = 1, \ldots, k\}$.

  The following algorithm assumes that $w \geq [k(k-1) + 1]L$.

  The idea is to perform every pair of comparisons in parallel, by subtracting the elements from one another. Parallel AND's are used to identify the position(s) of the maximum element.

  Algorithm:

  1 Concatenate k copies of the kL bit string a to get a':
  Multiply a by $(0^{kL-1}1)^k$.
  2 Create an expanded version of the reverse of a
  with $k-2$ blank fields between each original field:
  Shift a' right $(k-1)L$ positions.
  AND with the mask $(0^{(k-2)L}1^L)^k$.
  3 Replicate fields, so there are $k-1$ adjacent copies of each number:
  Multiply by $(0^{L-1}1)^{k-1}$.
  4 Perform all pairwise comparisons:
  OR with $1^{w-k(k-1)L}(10^{L-1})^{k(k-1)}$.
  Subtract a'.
  5 Determine which elements won all their comparisons:
  Shift right 1 bit.
  OR with the mask $1^{w-k(k-1)L}(101^{L-2})^{k(k-1)}$.
  Subtract $(01^{(k-1)L-1})^k$.
  6 Find the index of a field of length $(k-1)L$ beginning with 1

  The word at the end of step 5 is a string which contains 1 in the leftmost bit of each field containing the maximum value. It can be used to create a mask to select all such fields.

  Example ($k = 3$, $L = 2$):

										c_1	0	c_2	0	c_3
$\times$	0	1	0	0	0	0	0	1	0	0	0	0	0	1
	0	c_3	0	c_1	0	c_2	0	c_3	0	c_1	0	c_2	0	c_3
$\rightarrow$	0	$*$	0	$*$	0	c_3	0	c_1	0	c_2	0	c_3	0	c_1
$\wedge$	0	0	0	0	1	1	0	0	1	1	0	0	1	1
	0	0	0	0	0	c_3	0	0	0	c_2	0	0	0	c_1
										$\times$	0	1	0	1
	0	0	0	c_3	0	c_3	0	c_2	0	c_2	0	c_1	0	c_1
$\vee$	0	0	1	0	1	0	1	0	1	0	1	0	1	0
	1	1	1	c_3	1	c_3	1	c_2	1	c_2	1	c_1	1	c_1
$-$	0	c_3	0	c_1	0	c_2	0	c_3	0	c_1	0	c_2	0	c_3
	$*$	$*$	$b_{3,1}$	$*$	$b_{3,2}$	$*$	$b_{2,3}$	$*$	$b_{2,1}$	$*$	$b_{1,2}$	$*$	$b_{1,3}$	$*$
$\rightarrow$	0	$*$	$*$	$b_{3,1}$	$*$	$b_{3,2}$	$*$	$b_{2,3}$	$*$	$b_{2,1}$	$*$	$b_{1,2}$	$*$	$b_{1,3}$
$\vee$	0	0	1	0	1	0	1	0	1	0	1	0	1	0
	1	1	1	$b_{3,1}$	1	$b_{3,2}$	1	$b_{2,3}$	1	$b_{2,1}$	1	$b_{1,2}$	1	$b_{1,3}$
$-$	0	0	0	1	1	1	0	1	1	1	0	1	1	1
	1	1	d_3	$*$	$*$	$*$	d_2	$*$	$*$	$*$	d_1	$*$	$*$	$*$

Here $*$'s denote bits that we don't care about, c_i is an abbreviation for $c_{i,1}$,

$$b_{i,j} = \begin{cases} 1 & \text{if } c_i \geq c_j \\ 0 & \text{if } c_i < c_j. \end{cases}$$

and $d_i = \bigwedge_{j \neq i} b_{i,j}$, which is true if and only if $a_i = \max\{a_j \mid j = 1, \ldots, k\}$.

- **Select the maximum number from each group**
  Given a string of length $gk(k-1)L$

  $$a = 0^{(k-2)L} a_{1,1} \cdots 0^{(k-2)L} a_{1,k} 0^{(k-2)L} a_{2,1} \cdots 0^{(k-2)L} a_{2,k} \cdots 0^{(k-2)L} a_{g,1} \cdots 0^{(k-2)L} a_{g,k}$$

  consisting of g groups of k numbers (each of value less than 2^{L-1} and located in the rightmost L positions of a field of $(k-1)L$ bits), compute a string of length $gk(k-1)L$

  $$m = 0^{(k^2-k-1)L} m_1 \cdots 0^{(k^2-k-1)L} m_g$$

  containing the maximum number in each group (each located in the rightmost L positions of a field of $k(k-1)L$ bits).

  The idea is to find the maximum number from each group in parallel, using an algorithm very similar to finding the maximum field in a string.

  Algorithm:

  1 Concatenate the numbers of each group in reverse order to get
  $0^{(k-2)kL} a_{1,k} \cdots a_{1,1} 0^{k(k-2)L} a_{2,k} \cdots a_{2,1} \cdots 0^{k(k-2)L} a_{g,k} \cdots a_{g,1}$:
  Multiply a by $(0^{kL-1}1)^{k-1}$.
  AND with the mask $(0^{(k-1)L} 1^{(k-1)L} 0^{(k-1)(k-2)L})^g$ to get t.
  AND a with the mask $(0^{(k-2)L} 1^L 0^{(k-1)^2 L})^g$.
  Shift right kL positions.
  OR with t.

Shift right $(k^2-3k+1)L$ positions.

2 Replicate each group of k numbers $k-2$ times to get
$a' = (a_{1,k}\cdots a_{1,1})^{k-1}(a_{2,k}\cdots a_{2,1})^{k-1}\cdots(a_{g,k}\cdots a_{g,1})^{k-1}$:
Multiply by $(0^{kL-1}1)^{k-1}$

3 Replicate the numbers within string a each $k-2$ times to get
$(a_{1,1})^{k-1}\cdots(a_{1,k})^{k-1}(a_{2,1})^{k-1}\cdots(a_{2,k})^{k-1}\cdots(a_{g,1})^{k-1}\cdots(a_{g,k})^{k-1}$:
Multiply a by $(0^{L-1}1)^{k-1}$

4 Perform all pairwise comparisons:
OR with $(10^{L-1})^{k(k-1)g}$.
Subtract a'.
Subtract the string $[(0^{L-1}1)^0(0^L)^{k-1}(0^{L-1}1)^1(0^L)^{k-2}\cdots(0^{L-1}1)^{k-1}(0^L)^0]^g$.

5 Determine which elements won all their comparisons:
Shift right 1 position.
OR with the mask $(101^{L-2})^{k(k-1)g}$.
Subtract $(01^{(k-1)L-1})^{kg}$.
AND with the mask $(10^{(k-1)L-1})^{kg}$.

6 Select the maximum number from each group.
Shift right $(k-1)L-1$ positions.
Multiply by 1^{L-1}.
AND with a.

7 Move the maximum number in each group to the leftmost L positions in the field of $k(k-1)L$ bits belonging to its group.
Multiply by $(0^{L-1}1)^{k(k-1)}$
AND with the mask $(1^L0^{(k^2-k-1)L})^g$

8 Move the maximum number in each group to the rightmost L positions in the field of $k(k-1)L$ bits belonging to its group.
Shift right $(k^2-k-1)L$ positions.

Note that after step 6, each group contains exactly one nonzero number. Therefore at most two numbers are added together during the multiplication in step 7. Since the numbers are all less than 2^{L-1} and they are located in L bit fields, this addition causes no overflow from one field to another.