# Exam 2023 Dec

# Problems

## Problem 1: Big shelter - NP-hard

## Bit Shelter
### Problem ID: bitshelter

Recall that "or" of two bits is defined as

$$a \vee b = \begin{cases} 1 & \text{if } a = 1 \text{ or } b = 1 \text{ (or both)} \\ 0 & \text{otherwise,} \end{cases}$$

and that this concept is extended to length-$K$ bitstrings by defining $z = x \vee y$ as

$$z_i = x_i \vee y_i \qquad (1 \leq i \leq K)\,.$$

This is called the bitwise-or of $x$ and $y$. For instance, $1001 \vee 0011 = 1011$. It extends naturally to more than two strings (the operation is associative), for instance, $10011 \vee 00111 \vee 11101 = 11111$.

Given $N$ many bitstrings, each of length $K$, select as few as possible so that their bitwise-or is $1 \cdots 1$ (i.e., $K$ many 1s).

### Input

On the first line of input, the number $N$ of bitstrings, and the length $K$ of every bitstring. Each of the following $N$ lines contains a bitstring.

It is guaranteed that the bitwise-or of *all* the input bitstrings is $1 \cdots 1$ (i.e., $K$ many 1s.) You can assume that the bitstrings are all different.

### Output

Output $M$ lines, where $1 \leq M \leq N$. Each line must be a bitstring from the input. The bitwise-or of the $M$ lines must be $1 \cdots 1$ (i.e., $K$ many 1s), and $M$ must be minimal.

| Sample Input 1 | Sample Output 1 |
|---|---|
| 5 14<br>00011110001111<br>11111110000000<br>00000001111111<br>01100000110000<br>10000001000000 | 00000001111111<br>11111110000000 |

| Sample Input 2 | Sample Output 2 |
|---|---|
| 4 2<br>11<br>01<br>10<br>00 | 11 |

## Problem 2: Cluster - Graph traversal

# Cluster
## Problem ID: cluster

You are given a sequence of $N$ bitstrings, each of length $K$. Your task is the find a largest *cluster of ones*. Such a cluster consists of 1s that are adjacent to other 1s, in either of the four directions left, right, up, and down.

For instance, in Sample Input 1 there are two such clusters; of size 8 and 7, respectively. Note that Sample Input 2 contains five clusters, the largest is the one in the top-right corner.

## Input

On the first line, the number $N$ of bitstrings and the length $K$ of each bitstring.

Then follow $N$ lines, each containing exactly one bitstring.

## Output

A single integer: the number $r$ of 1s in the largest cluster in the input.

**Sample Input 1**

```
5 5
11111
11000
10000
00111
01111
```

**Sample Output 1**

```
8
```

**Sample Input 2**

```
4 4
1001
0101
0010
0001
```

**Sample Output 2**

```
2
```

**Sample Input 3**

```
5 5
11111
10001
10101
10001
11111
```

**Sample Output 3**

```
16
```

**Sample Input 4**

```
1 10
0000000000
```

**Sample Output 4**

```
0
```

# Problem 3: Decrease - Dynamic programming

## Decrease
### Problem ID: decrease

A bit string can be *decreased* into another bitstring of the same length by turning some of its 1s into 0s. For instance, 0110 can be decreased into 0100 and into 0010 (and even into 0000). On the other hand, 0110 cannot be decreased into 1000.

Given some bitstrings of equal length, select a maximal sequence such that each bitstring can be decreased into its successor.

### Input

On the first line of input, the number $N$ of bitstrings, and the length $K$ of every bitstring. Then follow $N$ lines, each containing a bitstring.

### Input

On the first line of input, the number $N$ of bitstrings, and the lengh $K$ of each bitstring. Then follow $N$ lines, each containing a bitstring. You can assume that they are all different.

### Output

Print $M$ lines of bitstrings, where $1 \leq M \leq N$. Each bitstring must occur in the input, and they must all be different. Moreover, for $1 \leq i < M$, the $i$th bitstring in the output can be decreased into the $(i + 1)$st. (In other words, if $x$ is followed by $y$ then $x_j \geq y_j$ for all $1 \leq j \leq K$.) The number of lines $M$ must be maximal.

If more than one solution of maximal size $M$ exists (such as in Sample Input 3), any will do.

**Sample Input 1**

```
5 5
00000
11000
00111
01000
11111
```

**Sample Output 1**

```
11111
11000
01000
00000
```

**Sample Input 2**

```
1 10
1101110111
```

**Sample Output 2**

```
1101110111
```

**Sample Input 3**

```
4 2
00
01
10
11
```

**Sample Output 3**

```
11
10
00
```

## Problem 4: Expensive Ones - Greedy

# Expensive Ones

## Problem ID: expensiveones

You are given $N$ many bitstrings of length $K$ and want to print out as many as you can. However, your printing shop is running out of 1s; there are only $R$ many 1s left. (On the other hand, there are way more than $N \cdot K$ many 0s left, so you won't run out of those.)

### Input

On the first line, the number $N$ of bitstrings, the length $K$ of every bitstring, and the number $R$ of remaining 1s in the printing shop. You can assume $0 \leq R \leq N \cdot K$. You can assume that the bitstrings are different.

### Output

Print $M$ lines, with $0 \leq M \leq N$, each of which is a single bitstring from the input. They have to be all different, their order does not matter, and the total number of 1s in those $M$ lines must be $\leq R$.

The number $M$ of printed bistrings must be as large as possible. If there is more than one valid solution, any will do.

**Sample Input 1**

```
6 5 4
01110
01010
11110
10000
00000
01100
```

**Sample Output 1**

```
10000
00000
01110
```

**Sample Input 2**

```
2 3 6
111
110
```

**Sample Output 2**

```
110
111
```

**Sample Input 3**

```
4 10 0
0010000000
0000001000
0000000000
0000000001
```

**Sample Output 3**

```
0000000000
```

**Sample Input 4**

```
1 5 1
00011
```

**Sample Output 4**

```
```

## Problem 5: To Xor or not to Xor - Flow

# To Xor Or Not To Xor?
## Problem ID: toxorornottoxor

We recall the definition of bitwise boolean operations on bitstrings: First recall the bit operations $\wedge$ (also known as "and", conjunction), $\vee$ (also known as "or", disjunction), and $\oplus$ (also known as "xor", "exclusive or") defined on single bits $a, b \in \{0, 1\}$ as follows:

| $a$ | $b$ | $a \vee b$ | $a \wedge b$ | $a \oplus b$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Then, for a pair of equal-length bitstrings $x_1 \cdots x_k, y_1 \cdots y_k \in \{0,1\}^k$ we define the result of $\vee$, $\wedge$, and $\oplus$ element-wise:

- the $i$th elment of $x \vee y$ is $x_i \vee y_i$.
- the $i$th elment of $x \wedge y$ is $x_i \wedge y_i$.
- the $i$th elment of $x \oplus y$ is $x_i \oplus y_i$.

For instance, if $x = 011$ and $y = 110$ then $x \vee y = 111$, $x \wedge y = 010$, and $x \oplus y = 101$.

You are given $N$ many pairs of bitstrings, each of length $k$. The task is to associate with each pair a single operation so as to produce the largest number $d$ of *different* bitstrings.

Consider for instance Sample Input 1 and three possible sets of associations:

We would construct $d = 2$ different bitstrings like this:

$$1110 \vee 1111 = 1111$$
$$1111 \vee 0001 = 1111$$
$$1010 \wedge 0101 = 0000$$

We would construct $d = 3$ different bitstrings like this:

$$1110 \vee 1111 = 1111$$
$$1111 \wedge 0001 = 0001$$
$$1010 \wedge 0101 = 0000$$

A worst possible answer would be this, which creates the same bitstring for each pair, so $d = 1$.

$$1110 \vee 1111 = 1111$$
$$1111 \vee 0001 = 1111$$
$$1010 \oplus 0101 = 1111$$

## Input

On the first line, the number $n$ of pairs of bitstrings, and the length $k$ of each bitstring. Then follow $n$ lines, each with a pair of bitstrings separated by space.

## Output

Output $n + 1$ lines. On the first line, the integer $d$ of how many *different* bistrings you will write. Then follow $n$ lines; the $i$th of these output lines contains either $\vee$, $\wedge$, or $\oplus$ of the $i$th input line. There must be exactly $d$ different lines among these $n$ lines, and $d$ must be maximal. (If there are several valid outputs, any of them will do.)

**Sample Input 1**

```
3 4
1110 1111
1111 0001
1010 0101
```

**Sample Output 1**

```
3
1111
0001
0000
```

**Sample Input 2**

```
3 2
11 11
11 00
00 11
```

**Sample Output 2**

```
2
11
00
00
```

# 1. Counterexample

Looking through the problems on pages 3–7, as soon as Gordon Gecko reads the description of problem "Decrease", he is sure it can be solved by a greedy algorithm. His rough idea is the following:
"First find the bitstring with the most 1s, say $x$, and add it to the solution.
Then consider all bitstrings that $x$ can be decreased into; among those pick the one with the most 1s.
Make that the new $x$; and repeat until no bitstrings are left."

It's clear that Gordon hasn't really thought a lot about data structures or running times, but you could probably fix that for him.
However, his idea is wrong on a more fundamental level.

## 1.a (3 pt.)

Give a small but complete instance (as a concrete bitstring) on which Gordon's algorithm is guaranteed to fail to find an optimal solution. Specify a non-optimal solution that would be found by Gordon's algorithm, and what an optimal solution would be instead.

## 1.a - Answer

Consider following input:

```
0 1 1 1 1
1 1 1 0 0
1 0 0 0 0
```

Gordon's algorithm would pick the first row as it has the most $1$s.
Then he is unable to reduce it to any other row.
Thus outputting:

```
0 1 1 1 1
```

This is the wrong answer.
The optimal solution is to select the second row first and then the third row.
Thus producing the following output:

```
1 1 1 0 0
1 0 0 0 0
```

Thus his solution produced $M = 1$ while the optimal solution is $M = 2$.

# 2. Greedy

One of the problems on pages 3–7 can be solved by a simple greedy algorithm.

## 2.a (1 pt.)

Which one?

## 2.a - Answer

"Expensive ones" can be solved by a greedy algorithm.

## 2.b (2 pt.)

Describe the algorithm, for example by writing it in pseudocode. (Ignore parsing the input.) You probably want to process the input in some order; be sure to make it clear *which* order this is (increasing or decreasing order of start time, alphabetic, colour, age, size, x-coordinate, distance, number of neighbours, scariness, etc.). In other words, don't just write "sort the input."

## 2.b - Answer

```
bit_strings = [b_1, b_2, ..., b_n]

// Sort so the bitstring with least 1s is first
// O(n log n)
bitstrings.sort_ascending_by_number_of_ones

remaining_1s = R
selected_bit_strings = {}

// O(n)
foreach bit_string in bit_strings {
  if (selected_bit_strings.contains(bit_string))
    continue
  else if (remaining_1s - bit_string.number_of_ones < 0)
    return selected_bit_strings

  remaining_1s -= bit_string.number_of_ones
  selected_bit_strings.add(bit_string)
}
```

## 2.c (1 pt.)

State the running time of your algorithm in terms of the input parameters. (It must be polynomial in the input size.)

## 2.c - Answer

The algorithm sorts the bitstrings according to the number of 1s in ascending order.
This takes $O(n \log n)$ time.
Then we have to iterate trough all the bitstrings, and identify if we can add them to the solution - which takes $O(n)$ time.

Thus if $n$ denotes the amount of bitstrings, the total running time will be:

$$O(n \log n)$$

This of course assumes that we can calculate the amount of 1s in each bitstring in constant time.
If this is not the case then let $m$ denote the length of the bitstrings, and the running time will be:

$$O(nm \log n)$$

## 3. Graph traversal

One of the problems on pages 3–7 can be efficiently solved using (possibly several applications of) standard graph traversal methods (such as breadth-first search, depth-first search, shortest paths, connected components, spanning trees, etc.), and without using more advanced design paradigms such as dynamic programming or network flows.

## 3.a (1 pt.)

Which one?

## 3.a - Answer

"Cluster" can be solved by graph traversal.

## 3.b (2 pt.)

Describe your algorithm. As much as you can, make use of known algorithms. (For instance, don't re-invent a well-known algorithm. Instead, write something like "I will use Blabla's algorithm [KT, p. 342] to find a blabla in the blabla.")

## 3.b - Answer

We can both use a DFS or BFS to solve this problem.
Let's use a modified BFS.
First let us go trough the 2D array and when we encounter the first 1 we will start a BFS from that point.
Then we will explore all the connected 1s and mark them as visited set (which needs to be global).
While this is done we will create a count how many 1s we encounter.
When no ones are left we will have to look at the rest of the 2D array (which has not been explored) and repeat the process.

Then lastly we can return the largest value or counter registered and then return that as the largest cluster we found.

## 3.c (1 pt.)

State the running time of your algorithm in terms of the parameters of the input.

## 3.c - Answer

The 2D array is bounded by $N$ and $K$.
Thus this can be seen as a graph which at most has $N \cdot K$ vertices and $N \cdot K$ edges.

Thus the running time of the algorithm will be:

$$O(N \cdot K)$$

# 4. Dynamic programming

One of the problems on pages 3–7 is solved by dynamic programming.

## 4.a (1 pt.)

Which one?

## 4.a - Answer

"Decrease" can be solved by dynamic programming.

## 4.b (3 pt.)

Following the book's notation, let OPT(…) denote the value of a partial solution.
(Maybe you need more than one parameter, like OPT(i, v).
Who knows?
Anyway, tell me what the parameters are—vertices, lengths, etc. and what their range is.
Use words like "where $i \in \{1, \dots, k^2\}$ denotes the length of BLABLA" or "where $v \in R$ is a red vertex".)
Give a recurrence relation for OPT, including relevant boundary conditions and base cases.
Which values of OPT are used to answer the problem?

## 4.b - Answer

Assume all bitstrings $B = \{b_1, b_2, \cdot, b_n\}$ where $b_i$ is a bitstring consisting of 0s and 1s.
Define a DAG $G = (V, E)$ where:

- $V = B$
- $(u, v) \in E$ if bitstring $u$ can be decreased to bitstring $v$.

$\mathrm{OPT}(b)$ denotes the largest sequence of bitstrings that can be selected.

$$OPT(b) = \begin{cases} 1 & \text{if } b \text{ has no outgoing edges} \\ 1 + \max_{(u,v) \in E} \mathrm{OPT}(v) & \text{otherwise} \end{cases}$$

Solution is thus:

$$\max_{b \in B} \mathrm{OPT}(b)$$

## 5. Flow

One of the problems on pages 3–7 is easily solved by a reduction to network flow.

### 5.a (1 pt.)

Which one?

### 5.a - Answer

"To Xor or not to Xor" can be solved by a reduction to network flow.

### 5.b (3 pt.)

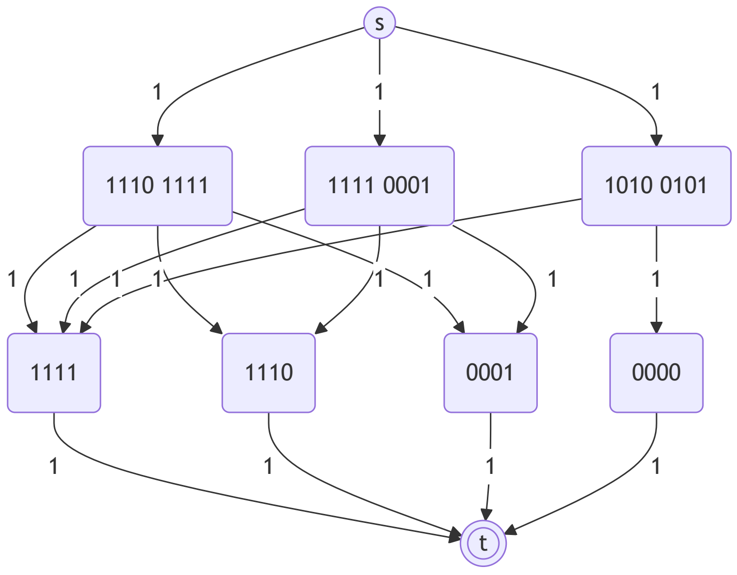
Explain the reduction.
Start by drawing the graph corresponding to Sample Input 1.
Be ridiculously precise about which nodes and arcs there are, how many there are (in terms of size measures of the original problem), how the nodes are connected and directed, and what the capacities are.
Describe the reduction in general (use words like "every node corresponding to a giraffe is connected to every node corresponding to a letter by an undirected arc of capacity the length of the neck").
What does a maximum flow mean in terms of the original problem, and what size does it have in terms of the original parameters?

### 5.b - Answer

To reduce the problem to a network flow problem we can do the following:

- Create a source node $s$
- Create a sink node $t$
- Create a node for each pair of bitstrings $b_i$
- For each $b_i$ create an arc of capacity 1 to each of the possible bitstrings $s_i$ that can be reached by using each operation
  - There must be no dublicate $s_i$ nodes (meaning no two $s_i$ can have the same value)
- Create an arc of capacity 1 from each $s_i$ to the sink node $t$
- Create an arc of capacity 1 from the source node $s$ to each $b_i$ node

This will result in the following graph from Sample input 1:



We can now $N$ flow trough the graph (denotes the amount of bitstrings).
If we get a flow of $N$ then we have found the a valid solution and can run a DFS on the residual graph to find the unique bitstrings that were selected.

## 5.c (1 pt.)

State the running time of the resulting algorithm, be precise about which flow algorithm you use. (Use words like “Using Krampfmeier–Strumpfnudel’s algorithm ((5.47) in the textbook), the total running time will be $O(r^{17}\log^3 \epsilon + \log^2 k)$, where $r$ is the number of frontonzes and $k$ denotes the maximal weight of a giraffe.”)

## 5.c - Answer

As we have to find the max-flow we can use Ford-Fulkerson.
Specifically we can use the Edmonds-Karp algorithm.
Edmonds-Karp runs in $O(VE^2)$.
Dinitz's algorithm further reduces this to $O(V^2E)$.
Furthermore we have to run a DFS on the residual graph to find the unique bitstrings that were selected.

In the worst case we have $N$ bitstrings pairs, $3N$ possible bitstring results, $2$ extra nodes for the source and sink.
This would result in a total of $4N+2$ nodes and also $4N+2$ edges.

Thus the running time of the algorithm will be:

$$O(N^2)$$

# 6. NP-hard

One of the problems on pages 3–7 is NP-hard.

## 6.a (1 pt.)

Which problem is it? (Let’s call it $P_1$.)

## 6.a - Answer

"Big shelter" is NP-hard.
Thus $P_1 = \text{Big shelter}.$

## 6.b (1 pt.)

The easiest way to show that $P_1$ is NP-hard is to consider another well-known NP-hard problem (called $P_2$). Which?

## 6.b - Answer

We can consider "Subset-sum" as the known NP-hard problem.
Thus $P_2 = \text{Subset-sum}.$

## 6.c (0 pt.)

Do you now need to prove $P_1 \leq_p P_2$ or $P_2 \leq_p P_1$?

## 6.c - Answer

We need to prove $P_2 \leq_p P_1$.
Thus that "Subset-sum" can be reduced to "Big shelter".

## 6.d (3 pt.)

Describe the reduction.
Do this both in general and for a small but complete example.
In particular, be ridiculously precise about what instance is **given**, and what instance is **constructed** by the reduction, the parameters of the instance you produce (for example number of vertices, edges, sets, colors) in terms of the parameters of the original instance, what the solution of the transformed instance means in terms of the original instance, etc.

For the love of all that is Good and Holy, please start your reduction with words like “Given an instance to BLABLA, we will construct an instance of BLABLA as follows.”

## 6.d - Answer

Given instance $S = \{s_1, s_2, \ldots, s_n\}$ and a target value $T$.
$s_i$ and $T$ are bitstrings of length $K$.

The goal is to select as few $s_i$ as possible such that the sum of the selected $s_i$s equals $T$.
Thus that the bitwise $\mathrm{OR}$ of the selected $s_i$s equal $T$.

Create instance of "Big shelter" as follows:

- For each $s_i \in S$ create a bitstring
- For $T$ create a bitstring of $T$ of $K$ many 1s
- For the sum function replace it with a bitwise OR function

**Example** $S = \{001, 010, 101\}, T = 111$

The optimal solution is to select $s_1$ and $s_3$ as $001 \vee 101 = 111$.