# First International
# Lightning Model and Solve Competition

Held at CP2013
Organizers: Peter J. Stuckey and Håkan Kjellerstrand
with the technical support of Farshid Hassani Bijarbooneh

### Methods

In submitting each solution you should add a `method` line of the form

```
% method = ≪string≫;
```

The `method` line is *optional*, just so we can collect statistics on the methods used by each team and on each problem. The method string can be *any string* of 50 characters or less, but we suggest at least naming the optimization technology, e.g. `cp`, `mip`, `ls` (local search), `asp` (answer set programming), `ga` (genetic algorithms), `bespoke` (bespoke solution), `hand` (by hand), perhaps with more info, e.g. `cp-gecode`.

## Mqueens

The famous *n-queens* problem is actually a simplification of the much more important *m-queens* problem, which is originally attributed to Adardhir I the founder of the Sassanid Empire. His mother Rodhagh challenged him to cover his empire with the minimum number of castles so that no place was not protected by cavalry within 1 week riding. Adardhir simplified this problem into the *m-queens* problem, based on the game of chess, which was becoming popular at the time, under the name *shatranj*. His son Shapur I is attributed with the first optimal solution on a standard 8×8 chessboard. The *m-queens* problem, generalizing Adardhir's problem, is to cover an $n \times n$ chess board with as few queens as possible so that no queen can take another and no more queens can be placed on the board without being taken.

The $i^{th}$ instance of the problem is defined by an input text file `mqueens_`$i$`.txt` with a single integer $n$. The output should be a text file `mqueens_`$i$`_`$t$`.dzn` where $t$ is your team number, in the format:

```
q = [list of n queen positions for each row or 0];
obj = o;
% method = ≪string≫;
```

where $o$ is the number of queens used on the board.

For example given $n = 5$ a solution might be

```
q = [2, 0, 5, 3, 1];
obj = 4;
% method = "cp-g12fd";
```

representing the solution

```
. Q . . .
. . . . .
. . . . Q
. . Q . .
Q . . . .
```

and a minimal solution might be

```
    q = [5,2,0,3,0];
    obj = 3;
```

representing the solution

```
. . . . Q
. Q . . .
. . . . .
. . Q . .
. . . . .
```

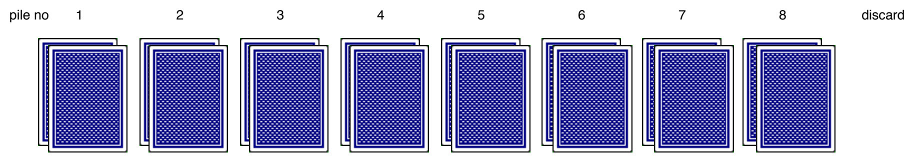
## Doubleclock

Clock patience is a very boring patience, since it is completely deterministic. The cards are dealt into 13 piles of 4 in the 12 hour positions, and the last pile (pile 13) in the center. The player begins by revealing the top card of the last pile, and placing it in the discard. If this card was number $j$ (A = 1, J = 11, Q = 12, K = 13) the player then reveals the top card of the $j^{th}$ pile and adds that to the discards. Using the number $j'$ on this card, the top card of the $j'^{th}$ pile is then revealed. This continues until either all cards are played to the discard, a winning game, or more likely, the last card indicates an empty pile.

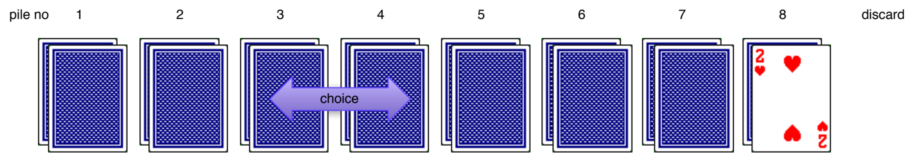
Double clock patience is a much more challenging game. There are two piles for each card number (of two cards), every time a card with number $j$ is revealed the player has a choice of which pile to choose from (or no choice if one pile is already empty). The play continues as before, with the player making binary choices. The player wins if all cards are played to the discard. Since double clock patience is very challenging one can also play to minimize restarts. When a card is revealed both of whose piles are empty, then the play restarts by taking the top card from the largest numbered non-empty pile. The aim of this game is to minimize the number of restarts.

We will play a generalized version of double clock patience with $n$ different card numbers and $2k$ suits. The cards are arranged in $2n$ piles of $k$ cards. The top card of the last pile is revealed. If its number if $j$ then we can choose to reveal the top card of pile $2j-1$ or $2j$. Play continues until either all piles are empty, or both piles $2j-1$ and $2j$ are empty. We then restart the game with the top card of the last non-empty pile.

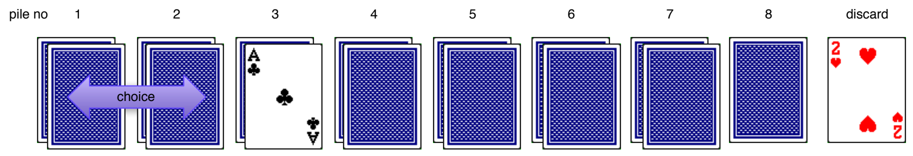
Below we illustrate the playing of the game for $n = 4$, $k = 2$. Initially there are 8 piles of 2 cards.



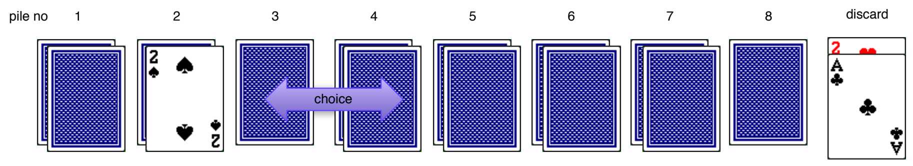
The top card of the last pile is revealed. Its a 2, so we can reveal the next card from pile 3 or 4.



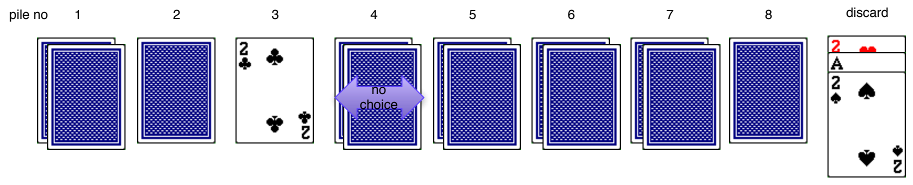
The 2 is moved to the discard and pile 3 is chosen to reveal. This shows a 1, so either piles 1 or 2 can be chosen next.



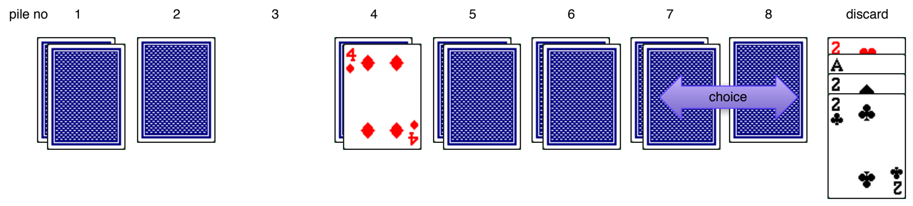
Pile 2 is chosen and another card 2 is revealed, again we can choose either pile 3 or 4.



Pile 3 is chosen, and another 2 is revealed and this time only pile 4 can be chosen.



The top card of pile 4 is revealed as a 4 and we can choose from pile 7 or 8.



The $i^{th}$ instance of the problem is defined by an input text file `doubleclock_`$i$`.txt` with two integers $n$ and $k$ and the initial layout of the cards as a permutation of the numbers $1..2nk$. Each integer $j$ represents the card with number $(j-1) \bmod n + 1$ and different suits. E.g. $2 = 2\heartsuit$, $6 = 2\diamondsuit$, $8 = 4\diamondsuit$, $9 = A\clubsuit$, $10 = 2\clubsuit$, $14 = 2\spadesuit$. The first $k$ cards in the sequence gives: pile 1 from bottom to top, the next $k$ cards give pile 2 from bottom to top, until the last $k$ cards are the last pile ($2n$) from bottom to top. The first card to be revealed is the last card in the sequence (the top of pile $2n$).

An example input file might be

```
4 2
16 5   1 14   10 9   12 8   6 7   11 15   13 4   3 2
```

which agrees with the sample game above.

The output should be a text file `doubleclock_`$i$`_`$t$`.dzn` where $t$ is your team number, in the format:

```
p = [order of discards, permutation of 1..2nk];
obj = ≪number of restarts≫;
% method = ≪string≫;
```

For example a correct output for the example game above would be:

```
p = [2, 9, 14, 10, 8, 3, 15, 11, 7, 6, 12, 4, 13, 1, 5, 16];
obj = 0;
% method = "mip";
```

## Smelt

Smelting is a complex process. A smelting factory can produce rectangular objects out of $m$ minerals using $r$ recipes. The smelting factory can only produce a maximum molten flow of $f_i, 1 \leq i \leq m$ ($\times$ 10kg) for each mineral each minute. Each recipe $k, 1 \leq k \leq r$ needs a component input flow of $c_{ki}, 1 \leq i \leq m$ for each mineral $i$, measured in the same 10kg units as the molten flows. A set of customer orders is a list of $n$ triples $t_j, h_j, w_j$ where $t_j \in 1..r$ is the recipe, $h_j$ is the height of the piece in metres, and $w_j$ is the width in metres. The factory has $l$ smelting lines which can produce 1 square metre of any recipe every minute, each line processing different orders simultaneously. An important rule that can't be violated is that all orders using the same recipe must be produced one after the other on the same smelting line, so that the recipe is cooked exactly once. There are complex interactions between the different recipes, and the production manager records these as $p$ production rules for determining the operations of the factory. Production rules can be of 4 types:

- type 1: all orders for recipe $k_1$ must be finished at least $d$ minutes before and order for recipe $k_2$ starts;
- type 2: at least one order for recipe $k_1$ cannot finish more than $d$ minutes before some order for recipe $k_2$ starts;

- type 3: at least one order for recipe $k_1$ must start no more than $d$ minutes after any order for recipe $k_2$ starts; and
- type 4: all orders for recipe $k_1$ must be finished no more than $d$ minutes after some order for recipe $k_2$ finishes.

Unfortunately the production manager is often overzealous in demanding productions rules, such that the foreman is unable to work out a way of completing a set of orders. In practice the foreman ignores some of the production rules in order to finish the orders. The aim is to minimize the number of ignored production rules while completing the orders in the minimum amount of time.

The $i^{th}$ instance of the problem is defined by an input text file `smelt_`*i*`.txt` with integers $m$ giving the number of minerals, then $m$ integers $f_i$ for each flow rate. Next is an integer $r$ giving the number of recipes, and then $r$ sets of $m$ integers defining $c_{ki}$ for each $1 \leq k \leq r$. Next is an integer $n$ giving the number of orders, followed by $n$ sets of three integers: $t_j, h_j, w_j$ for each order. Next an integer $l$ giving the number of smelting lines. Then an integer $p$ giving the number of production rules, and then 4 integers for each rule first the rule type (1–4) and then $k_1, d, k_2$ defining the data of the rule.

The output should be a text file `smelt_`*i*`_`*t*`.dzn` where $t$ is your team number, in the format:

```
s = [start time for each order];
e = [end time for each order];
l = [production line for each order];
obj = ≪1000 * number of violated production rules +
                maximum end time of any order≫;
% method = ≪string≫;
```

For example the input file

```
2   10 5
3   4 4  3 1  6 3
4   1 2 3  2 4 1  1 2 2  3 1 1
2
3   1 1 0 2   1 2 4 1   4 3 0 2
```

defines an instance with 2 minerals, 3 recipes, 4 orders, 2 smelting lines, and 3 production rules where $(type, k_1, d, k_2)$ are $(1,1,0,2)$, $(1,2,4,1)$ and $(4,3,0,2)$ respectively.

Example output might be

```
s = [4,10,0,13];
e = [10,14,4,14];
l = [2,2,2,1];
obj = 1014;
% method = "ls";
```

There may be no solution to a smelting problem in which case the expected output is simply

```
obj = -1;
```

## Seatplan

One of the most stressful decisions made by any young couple is how to arrange the seating at their wedding. The seating must satisfy a number of rules, such as alternating male and female positions on the tables and care must be taken to separate people from people they dislike or strongly dislike. The seatplan optimization problem is given $m$ male guests numbered $-m$ to $-1$ and $f$ female guests numbered 1 to $f$ and $t$ circular tables each with $s$ chairs to assign each guest to one of the $t \times s$ seats (seats 1..s are on table 1, seats s+1..2s are on table 2, etc) such that no male sits next to a male, and no female sits next to a female. The remaining constraints are soft, the aim is to minimize the number of violations.

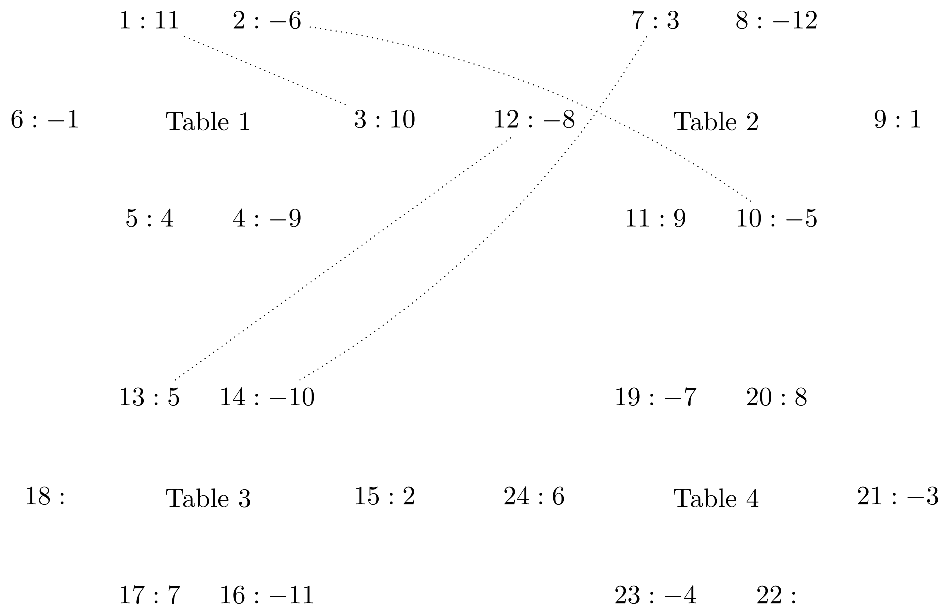
- Partners should be adjacent to each other
- Pairs of people who dislike each other should not be adjacent
- Pairs of people who strongly dislike each other should not be on the same table

The $i^{th}$ instance of the problem is defined by an input text file `seatplan_`$i$`.txt` with integers $m$ and $f$ defining number of males and females, next are integers $t$ and $s$ defining the number of tables and seats per table. Next are the number of partnerships $p$ and then $p$ pairs of integers representing the partnerships. Next, similarly, are the number of dislikes $d$ and then $d$ pairs of integers representing the pairs who dislike each other. Finally the number of strong dislikes $sd$ and then $sd$ pairs of integers representing the pairs who strongly dislike each other. For example the following input

```
12 11
4 6
8  -12 1  -11 2  -10 3  -9 4  -8 5  -7 6  -6 -5  11 10
20  -12 -11  -12 -8  -12 -5  -11 -1  -11 3  -10 -6  -9 7  -8 10  -7 5
    -6 4  -5 3  -4 -1  -3 6  -2 6  -1 9  1 -9  2 -9  3 -10  11 -7  10 -7
10  -12 2  -9 3  -8 5  3 6  4 -5  5 10  6 -10  8 -9  10 -4  11 -4
```

represents a problem with 12 males, 11 females, 4 tables of 6 chairs each, 8 partnerships, 20 dislikes and 10 strong dislikes.

A possible seating plan for this problem is shown below, where $sn : p$ shown seat number $sn$ with the person code $p$, and some seats are empty.



The violated soft constraints, in this example all partner constraints, are shown as arcs.

The output should be a text file `seatplan_`$i$`_`$t$`.dzn` where $t$ is your team number, in the format:

> `male = [`seat number for each male from $-m$ to $-1$ in order`];`
> `female = [`seat number of each female from 1 to $f$ in order`];`
> `obj = `≪number of violated soft constraints≫`;`
> `% method = `≪*string*≫`;`

For example the output for the solution shown above would be (remember `method` is optional)

```
male = [8, 16, 14, 4, 12, 19, 2, 10, 23, 21, 18, 6];
female = [9, 15, 7, 5, 13, 24, 17, 20, 11, 3, 1];
obj = 4;
```

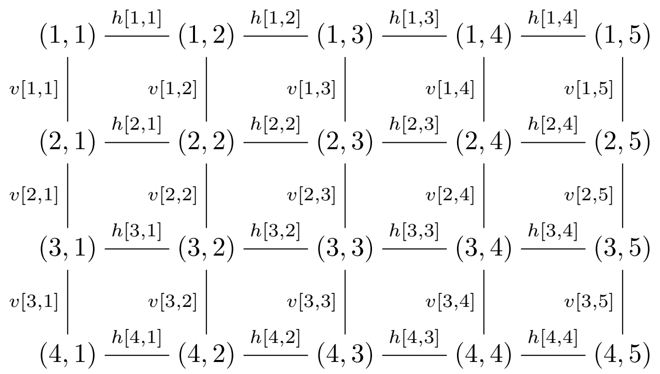
There may be no solution to a seat plan problem in which case the expected output is simply

```
obj = -1;
```

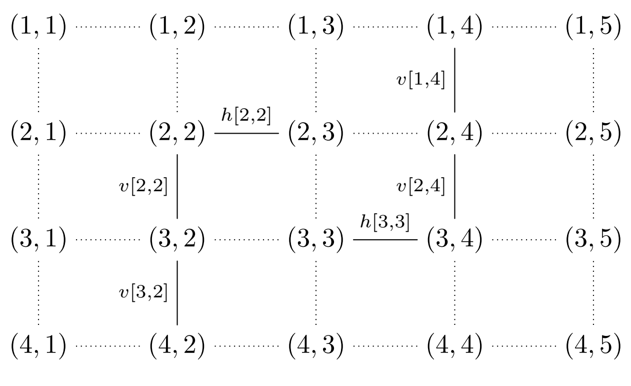
## Powerup

Routing is a complex combinatorial problem, but we will consider a very simple case. A circuit board is considered as a grid of positions of height $h$ and width $w$. The top line ($height = 1$) of the circuit board is the ground plane, and the bottom line ($height = h$) is the power plane. The problem is given a list $G$ of positions $(i, j)$ on the circuit board to be connected to the ground plane, and a list $P$ of positions $(i, j)$ to be connected to the power plane, find the minimum number of links in the grid that need to be filled to create the power configuration. Note that the ground plane cannot be connected to the power plane.

Consider an example circuit of height 4 and width 5, the possible horizontal links $h[i, j]$ and vertical links $v[i, j]$ represented as shown below.



Given the set $G = \{(3, 3), (3, 4)\}$ and $P = \{(2, 2), (2, 3)\}$ the unique minimal solution is using 6 links.



There may be no solution to a powerup problem, for example if we extend $P$ to be $P = \{(2, 1), (2, 2), (2, 3), (2, 4), (2, 5)\}$

The $i^{th}$ instance of the problem is defined by an input text file `powerup_`$i$`.txt` with integers $h$ and $w$ defining the height and width, then and integer $g = |G|$ giving the number of positions to be grounded then $g$ pairs of integers giving the $(i, j)$ positions in $G$, then an integer $p = |P|$ giving the number positioned to be powered, and then $p$ pairs of integers giving the $(i, j)$ positions in $P$. For example the input for the problem defined above would be

```
4 5
2   3 3  3 4
2   2 2  2 3
```

The output should be a text file `powerup_`$i$`_`$t$`.dzn` where $t$ is your team number, in the format:

> `hor = [`list of $h[i, j], i \in [1..h], j \in [1..w-1]$ in order`];`
> `ver = [`list of $v[i, j], i \in [1..h-1], j \in [1..w]$ in order`];`
> `obj = `≪number of used links≫`;`
> `% method = `≪*string*≫`;`

For example the output for the solution shown above would be

```
hor = [0,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,0];
ver = [0,0,0,1,0, 0,1,0,1,0, 0,1,0,0,0];
obj = 6;
% method = "bespoke";
```

Note that the extra spaces in the solution are only shown for clarity, they are not required.

For the input below which sets $P = \{(2, 1), (2, 2), (2, 3), (2, 4), (2, 5)\}$

```
4 5
2   3 3  3 4
4   2 1  2 2  2 3  2 4  2 5
```

the expected output is simply

```
obj = -1;
```

## Zamkeller

The study of sequences is a critical part of number theory. Many difficult sequences are only known for the very early elements in the sequence, see The On-Line Encyclopedia of Integer Sequences `oeis.org`. One little studied sequence is the *Zamkeller sequence*. Given the numbers $1..n$ and $1 < k < n$, the *Zamkeller[n,k] number* of a permutation $p$ is the minimum number of differential alternations of any of the subsequences of $p$ of the numbers divisible by $i = 1..k$. A sequence $p$ is a Zamkeller[$n$,$k$] permutation if it has the maximum Zamkeller[$n$,$k$] number for any permutation over $1..n$. The *Zamkeller[k]* sequence is the sequence of maximum Zamkeller[$n$,$k$] numbers as $n$ increases.

The differential alternations in a sequence of different integers is the number of times the difference in adjacent elements changes sign. The aim of this problem is given $n$ and $k$ to find a permutation of $1..n$ with highest Zamkeller[$n$,$k$] number.

Consider the permutation of 1..20 given by

1, 19, 15, 13, 5, 4, 8, 9, 17, 20, 12, 7, 3, 16, 2, 14, 6, 18, 11, 10

The number of differential alternations is 9. The subsequences for 2 and 3 are respectively

4, 8, 20, 12, 16, 2, 14, 6, 18, 10

with 7 alternations, and

15, 9, 12, 3, 6, 18

also with 3 alternations. The Zamkeller[20,3] number of this permutation is 3.

The $i^{th}$ instance of the problem is defined by an input text file `zamkeller_`$i$`.txt` with two integers $n$ and $k$. The output should be a text file `zamkeller_`$i$`_`$t$`.dzn` where $t$ is your team number, in the format:

```
p = [permutation of 1..n];
obj = ≪Zamkeller[n,k] number of permutation p≫;
% method = ≪string≫;
```

For example for the sample permutation shown the output would be

```
p = [1, 19, 15, 13, 5, 4, 8, 9, 17, 20, 12, 7, 3, 16, 2, 14,
6, 18, 11, 10];
obj = 3;
% method = "cp";
```