# MiniZinc

## *crashcourse tutorial*

# MiniZinc

MiniZinc è un linguaggio di modellizzazione sviluppato da NICTA in collaborazione con Univ. di Melbourne/Monash.

I modelli MiniZinc possono essere risolti con constraint programming o con risolutori MIP.

E' un sottinsieme di un linguaggio più potente e complesso: Zinc, rilasciato nel 2010.

Web page

http://www.minizinc.org/

Risorse on-line

http://www.minizinc.org/ide/index.html

http://www.minizinc.org/2.0/index.html

http://www.hakank.org/minizinc/

# MiniZinc: funzionamento



MiniZinc interpreta un modello e i suoi dati e li traduce in un modello di più basso livello: FlatZinc

- Il tool ***mzn2fzn*** implementa questa traduzione.
- `mzn2fzn file.mzn data.dzn`
  - crea file.fzn
- L'interprete FlatZinc elabora i file FlatZinc
  - output molto semplice (solo valori di variabili)
- MiniZinc legge l'output semplice e calcola quello presentato

# MiniZinc e Mzn

`minizinc` è un interprete minizinc

- Più vecchio
- Più stabile
- Affidabile per il risolutore FD

mzn (es mzn-g12fd) è uno script che usa mzn2fzn/flatzinc

- Usa mzn2fzn per convertire MiniZinc in FlatZinc
- Lancia l'interprete FlatZinc
- Prende l'output di FlatZinc e lo passa a MiniZinc
- Meno stabile, usa tutti i risolutori FlatZinc.

# Un primo modello (MIP)

Problema:

max $25\,B + 30\,T$

s.t. $(1/200)\,B + (1/40)\,T \leq 40$

$0 \leq B \leq 6000$

$0 \leq T \leq 4000$

```
var 0.0..6000.0: B;
var 0.0..4000.0: T;
constraint (1.0/200.0)*B+(1.0/140.0)*T <= 40.0;
solve maximize 25.0*B + 30.0*T;
output  ["B = ", show(B), " T = ", show(T), "\n"];
```

Si salva il modello in un file *.mzn poi (se minizinc nel path):

```
mzn-g12mip primo.mzn
```

# Secondo modello (FD)

```
% Colouring central Italy using nc
  colours
int: nc = 4;

var 1..nc: toscana; var 1..nc: marche;
var 1..nc: umbria; var 1..nc: lazio;
var 1..nc: abruzzo; var 1..nc: molise;

constraint toscana != umbria;
constraint toscana != marche;
constraint toscana != lazio;
constraint umbria != marche;
constraint umbria != lazio;
constraint marche != lazio;
constraint lazio != abruzzo;
constraint lazio != molise;
constraint marche != abruzzo;
constraint abruzzo != molise;

solve satisfy;
```

```
output ["toscana=", show(toscana),
  "\t umbria=", show(umbria), "\t
  marche=", show(marche), "\n",
"lazio=", show(lazio), "\t
  abruzzo=", show(abruzzo), "\t
  molise=", show(molise), "\n"];
```

# Secondo modello (FD)

Il modello è a dominio finito, si risolve con

```
>mzn-g12fd centro_italia.mzn
```

Il risultato è:

```
toscana=2  umbria=1   marche=3
lazio=4    abruzzo=1  molise=2
----------
```

Il modello MiniZinc deve avere suffisso .mzn

# Parametri

In MiniZinc ci sono due tipi di variabili, i parametri e le variabili decisionali.

*Parametri*: sono come le variabili nei normali linguaggi di programmazione.
Gli si *assegna esplicitamente un valore*.

Sono dichiarati *associati a un tipo* (o a un range/set).

Possono essere (opzionalmente) anche introdotti da **`par`**.

Si può scrivere, equivalentemente:

```
int: i=3;
par int: i=3;
int: i;  i=3;
```

# Variabili decisionali

Variabili decisionali: sono come le variabili matematiche. Sono dichiarate di un tipo con la parola chiave `var`.

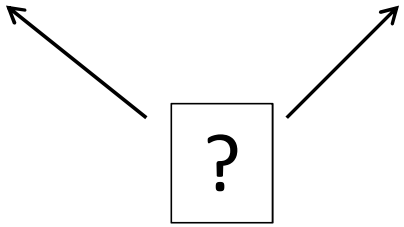
Il loro valore è calcolato da MiniZinc in accordo al modello.

Normalmente sono *dichiarate usando un range o un set* piuttosto che solo con un tipo.

Il range o set definisce il *dominio della variabile*.

Si può scrivere equivalentemente:

```
var 0..4: i;
var {0,1,2,3,4}: i;
var int: i; constraint i >= 0; constraint i <= 4;
```



# Tipi

I tipi permessi per le variabili sono:

- Integer: **`int`** o range **`1..n`** o **`set of int`**
- Floating point: **`float`** o range **`1.0 .. n.0`** o **`set of float`**
- Boolean: **`bool`**
- Stringhe: **`string`** (*non possono essere variabili decisionali*)
- **`Array[`*`range`*`] of `*`type`**
- **`Set of `*`type`**

Le variabili devono essere istanziate specificando se sono parametri o variabili decisionali.

Il tipo + istanziazione è detto ***type-inst***.

# Stringhe

Le stringhe servono solo per l'output

Un elemento di output ha la forma:

```
output <lista di stringhe>;
```

Le stringhe sono come in C, racchiuse fra “ “

Non possono essere più lunghe di una riga.

I caratteri speciali sono introdotti da backslash (\n \t … )

Funzioni predefinite:

- **`show(v)`**
- **`”bella” ++ ”stella”`** per concatenazione

# Espressioni aritmetiche

Operatori aritmetici standard:

- Float: **`*  /  +  -`**
- Integer: **`*  div mod  +  -`**

Operatori aritmetici relazionali:

**`==  !=  >  <  >=  <=`**

Non c'è casting automatico da integer a float

Il casting deve essere fatto esplicitamente con la funzione **`int2float(`*`intexp`*`)`**

# File di dati

I dati di input a un modello possono essere caricati da console o da un file di dati. I file dati devono avere estensione .dzn

Esempio: modello su calcolo di bilanci di un mutuo:

```
% variabili
var float: R; % rateo trimestrale
var float: P; % ammontare ricevuto
var 0.0 .. 100.0: I; % tasso di interesse
%
var float: B1; % bilancio primo trimestre
var float: B2; % bilancio secondo trimestre
var float: B3; % bilancio terzo trimestre
var float: B4; % bilancio finale
```

```
constraint B1 = P * (1.0 + I) – R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output …
```

Per poter valutare scenari diversi sulle variabili R, A, I, B1, ..,B4 si può fare l'inizializzazione da data file

- sx: prestito 1000€ al 4% rata € 260
- dx: prestito 1000€ al 4% ripagando tutto

```
I = 0.04;
P = 1000.0;
R = 260.0;
```

```
I = 0.04;
P = 1000.0;
B4 = 0.0;
```

Il modello viene richiamato così

```
> mzn-g12mip loan.mzn loan1.dzn
```

# Struttura di un modello 1

Un modello MiniZinc consiste di una sequenza di elementi.
L'ordine in cui appaiono è ininfluente.

Gli elementi possibili sono

- Elementi di inclusione
  **`include <`*`filename (a string literal)`*`>;`**
- Elementi di output
  **`output <`*`lista di stringhe`*`>;`**
- Dichiarazione di variabili
- Assegnamento di variabili
- Vincoli
  **`constraint <`*`espressione booleana`*`>;`**

(continua …)

# Struttura di un modello 2

Elementi possibili (cont.)

- elemento *solve* (una e una sola volta), può essere:

```
solve satisfy;
solve maximize <espressione aritmetica>;
solve minimize <espressione aritmetica>;
```

- Predicati e test
- Annotazioni

I nomi delle variabili cominciano con una lettera, seguita da lettere, underscore o cifre

L'underscore '_' da solo denota una variabile decisionale anonima.

# Esercizio

Sappiamo fare due tipi di torte.

| **Torta di banane** | **Torta al cioccolato** |
|---|---|
| 250g farina con lievito, | 200g farina con lievito, |
| 2 banane, | 75g cacao, |
| 75g zucchero | 150g zucchero |
| 100g burro | 150g burro. |

Abbiamo 4kg di farina, 6 banane, 2kg di zucchero, 500g di burro e 500g di cacao.

**Esercizio**: scrivere un modello MiniZinc che determina quante torte di ciascun tipo dovremmo fare per massimizzare il nostro profitto, sapendo che possiamo vendere una torta al cioccolato per €4.50 e una di banane per €4.00.

(v. manuale minizinc)

# Set

I set si dichiarano così:

**`set of` *`type`***

Sono ammessi solo con integer, float o Booleani.

Espressioni:

- Set di variabili: **`{e1,...,en}`**
- I range di integer o di float sono set
- Operatori predefiniti:

  **`in, union, intersect, subset, superset, diff, symdiff`**
- La cardinalità di un set è data da **`card`**

Esempi:

```
set of int: products = 1..nproducts;
{1,2} union {3,4}
```

I set possono essere usati come tipi.

# Array

Un array, mono o multi-dimensionale, si dichiara così

**`array[`*`index_set1,index_set2,  …,`*`] of`** ***`type`***

L'index set di un array può essere

- un *range* di interi
- Il *nome di una variabile* che rappresenta un set di interi.

Gli elementi di un array possono essere qualunque cosa tranne un altro array

Possono anche essere variabili decisionali.

Esempio:

```
array[products, resources] of int: consumption;
array[products] of var 0..mproducts: produce;
```

La funzione **`length`** ritorna il numero di elementi di un array 1D.

# Array

array 1-D possono essere inizializzati tramite una lista

```
profit = [400, 450];
capacity = [4000, 6, 2000, 500, 500];
```

array 2-D sono inizializzati con una lista contenente "|" per separare le righe

```
consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

Array di qualunque dimensione *(in pratica ≤ 3)* possono essere inizializzati con la famiglia di funzioni array*n*d :

```
consumption= array2d(1..2,1..5,
  [250,2,75,100,0,200,0,150,150,75];
```

Con array 1D si può usare l'operatore di concatenazione ++:

```
profit = [400]++[450];
```

# Pianificazione produzione



Probabilmente nella soluzione ricette e ingredienti sono stati inseriti nel codice.

Le ricette sono un semplice esempio di pianificazione della produzione in cui si desidera:

- determinare quanto produrre di ogni prodotto per massimizzare il profitto finale
- utilizzare nella produzione un quantitativo di risorse non superiore al massimo disponibile.

Si può scrivere un modello MiniZinc generico per questo problema.

# Pianificazione produzione



```
% Number of different products
int: nproducts;
set of int: products = 1..nproducts;

%profit per unit for each product
array[products] of int: profit;

%Number of resources
int: nresources;
set of int: resources = 1..nresources;

%amount of each resource available
array[resources] of int: capacity;

%units of each resource required to
produce 1 unit of product
array[products, resources] of int:
consumption;

% bound on number of products
int: mproducts = max (p in products)
      (min (r in resources where
consumption[p,r] > 0) (capacity[r] div
consumption[p,r]));

% Variables: how much should we make of
each product
array[products] of var 0..mproducts:
produce;

% Production cannot use more than the
available resources:
constraint forall (r in resources) (
    sum (p in products) (consumption[p,
r] * produce[p]) <= capacity[r]
);

% Maximize profit
solve maximize sum (p in products)
(profit[p]*produce[p]);

output [ show(produce)];
```

Notare come si siano utlizzati array e set.

# Comprehensions

MiniZinc implementa le *comprehension* per specificare insiemi di valori ammissibili

set comprehension:

{ expr | generator 1, generator 2, …}
{ expr | generator 1, generator 2, … where bool-expr }

array comprehension:

[ expr | generator 1, generator 2, …]
[ expr | generator 1, generator 2, … where bool-expr ]

Esempio

**`{i+j | i,j in 1..3 where i<j}`** = {1+2, 1+3, 2+3} = {3,4,5}

**Esercizio**: a cosa corrisponde b?

```
set of int: cols = 1..5;
set of int: rows = 1..2;
array [rows,cols] of int: c= [| 250,2,75,100,0, | 200,0,150,150,75 |];
b = array2d(cols, rows, [c[j,i] | i in cols, j in rows]);
```

# Iterazioni

MiniZinc fornisce diverse funzioni per iterare su una lista o un set:

- Lista di numeri: **`sum, product, min, max`**
- Lista di vincoli: **`forall, exists`**

MiniZinc accetta una sintassi speciale per le iterazioni:

Es,

```
forall (i,j in 1..10 where i<j) (a[i] != a[j]);
```

è equivalente a

```
forall ([a[i]!=a[j] | i,j in 1..10 where i<j]);
```

# Asserzioni

Permettono di controllare la validità dei valori delle variabili.

Funzione booleana **`assert(`*`boolexp,stringexp`*`)`** restituisce true se boolexp è vera, sennò stampa stringexp ed esce.

Può essere usata in un constraint. Es.:

```
int: nres;
constraint assert(nres > 0, "Error: nres =< 0");

array[resources] of int: capacity;
constraint assert(
   forall(r in resources)(resources[r] >= 0),
   "Error: negative capacity");
```

**Esercizio**: controllare che tutti gli elementi di consumo siano non negativi

array[prodotti, risorse] of int: consumo;

# If-then-else

**`if <boolexp> then <exp> else <exp> endif`**

Es.:

```
if y != 0 then x / y else 0 endif;
```

L'espressione booleana ***non può contenere variabili decisionali***, solo parametri.

Solo all'interno di un elemento **`output`**, si può usare la funzione predefinita **`fix`** che controlla che il valore di una variabile decisionale sia stato fissato e converte la variabile decisionale in un parametro.

# Constraint

Il cuore di un modello MiniZinc

Un constraint può essere una qualunque espressione booleana

I letterali Booleani sono

**`true, false`**

Operatori Booleani

**`/\  \/  <-  ->  <->  not`**

(and, or, solo-se, implica, sse, not)

Vincoli globali, es.: **`alldifferent`**

# Constraint: esempio



Problema di scheduling in cui si ha un insieme di task e una sola risorsa.

start[i] e duration[i] definiscono istante di inizio e durata di ogni task i

Per garantire che le attività non si sovrappongano

```
constraint forall (i,j in tasks where i != j)
           (  start[i] + duration[i] <= start[j] \/
              start[j] + duration[j] <= start[i] );
```

# Array Constraint



L'indice i di accesso ad un elemento (a[i]) *può essere una espressione che contiene variabili decisionali*, definendo quindi *implicitamente un vincolo sull'array*.

Esempio, problema del ***matrimonio stabile***.

Ci sono *n* donne e *n* uomini.

Ogni uomo definisce una lista ordinata delle donne (1 preferita, n temutissima) e viceversa.

Si vuole definire il marito / moglie di ogni donna / uomo col vincolo che i matrimoni siano stabili, cioè:

- Se *m* preferisce un'altra donna *a* a sua moglie *w*, *a* preferisce suo marito a *m*
- Se *w* preferisce un altro uomo *a* a suo marito *m*, *a* preferisce sua moglie a *m*

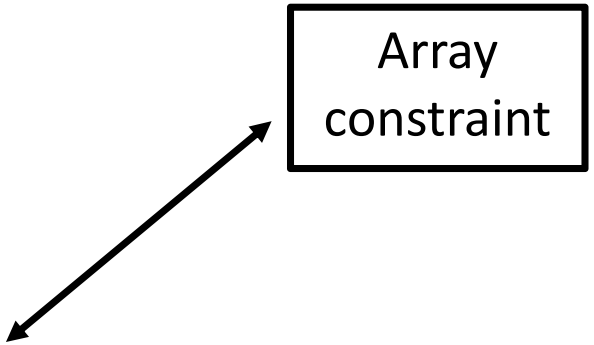
# Matrimonio stabile



```
int: n;

array[1..n,1..n] of int: rankWomen;
array[1..n,1..n] of int: rankMen;

array[1..n] of var 1..n: wife;
array[1..n] of var 1..n: husband;
% vincoli di assegnamento
constraint forall (m in 1..n) (husband[wife[m]]=m);
constraint forall (w in 1..n) (wife[husband[w]]=w);
```

**Esercizio**: qui i vincoli di stabilità

```
solve satisfy;
output ["wives= ", show(wife),"\n", "husbands= ", show(husband)];
```



# Higher-order constraint



La funzione di casting predefinita ***bool2int*** (falso → 0, vero → 1) permette di utilizzare i cosiddetti *higher order constraint*:

***Problema della sequenza magica***: trovare una lista di numeri $S = [s_0,\ldots,s_{n-1}]$ tale che $s_i$ sia il numero di volte in cui il numero $i$ appare in $S$.

```
int: n;
array[0..n-1] of var 0..n: s;

constraint
 forall(i in 0..n-1) (
   s[i] = sum(j in 0..n-1)(bool2int(s[j]=i)));

solve satisfy;
```

# Set Constraint



MiniZinc permette di utilizzare i set di integer come variabili decisionali

Problema ***knapsack 0/1***

```
int: n;
int: capacity;

array[1..n] of int: profits;
array[1..n] of int: weights;

var set of 1..n: knapsack;

constraint sum (i in knapsack) (weights[i]) <= capacity;

solve maximize sum (i in knapsack) (profits[i]) ;

output [show(knapsack)];
```

***Non funziona***: non si può iterare su set di variabili

**Esercizio** (ma v. dopo): riscrivere il modello in modo che non iteri sul var set

# Tipi enumerativi



I tipi enumerativi (v. c#) servono per utilizzare insiemi di oggetti oggetti, vengono gestiti come interi

```
enum people = {anna, bea, carlo, david};
```

E' equivalente a

```
set of int: people = 1..4;
int: anna  = 1;
int: bea   = 2;
int: carlo = 3;
int: david = 4;
array[people] of string: nomi = ["anna", "bea", "carlo", "david"];
```

# Esercizio 1: Magic Square



Un quadrato magico di lato *n* è una matrice quadrata di numeri di *n* righe e *n* colonne, tale che la somma degli elementi di ogni riga, colonna e diagonale della matrice abbia lo stesso valore.

Esempio di un quadrato magico 3×3:

2 7 6

9 5 1

4 3 8

**Esempio:** scrivere un modello MiniZinc che generi quadrati magici di ordine *n*

# Magic square



Il problema del quadrato magico può essere visto come un CSP in cui:

Variabili: gli elementi della matrice

Domini: 1..N*N

Vincoli:

- Somma magica= somma delle colonne = somma delle righe = somma delle diagonali
- Eliminazione simmetrie
- Vincolo ridondante:

  ***Somma magica = $N(N^2+1)/2$***

# Magic square

```
include "globals.mzn";

int: n = 4;

int: total = ( n * (n*n + 1)) div 2;
array[1..n,1..n] of var 1..n*n: magic;

solve satisfy;

constraint
        all_different([magic[i,j] | i in 1..n, j in 1..n])
        /\
        forall(k in 1..n) (
           sum(i in 1..n) (magic[k,i]) = total
           /\
           sum(i in 1..n) (magic[i,k]) = total
        )
       /\ % diagonal
       sum(i in 1..n) (magic[i,i]) = total
       /\ % diagonal
       sum(i in 1..n) (magic[i,n-i+1]) = total
;

output [ "Total: " ++ show(total) ++ "\n" ] ++ [  %   show(magic)...
```

# Exercizio 2: Task Allocation



Avendo

- Un insieme di attività, tasks
- Un insieme di operatori, workers
- Per ogni operatore, l'insieme di attività che è qualificato a svolgere
- Il costo di ogni operatore

**Esercizio:** scrivere un modello MiniZinc che trova il sottinsieme di operatori che può svolgere tutte le attività dell'insieme a costo minimo

# Criptoaritmetica



Problema: trovare le cifre che sostituite alle lettere soddisfano l'addizione:
SEND + MORE = MONEY

Un modello CS:

Variabili: {S,E,N,D,M,O,R,Y}

Domini: {0,1,2,3,4,5,6,7,8,9}

Vincoli:

- Variabili tutte diverse
- S!=0, M!=0
- $S*10^3+E*10^2+N*10+D$ {SEND}
  $+ M*10^3+O*10^2+R*10+E$ {MORE}
  $= M*10^4+ O*10^3+N*10^2+E*10+Y$ {MONEY}

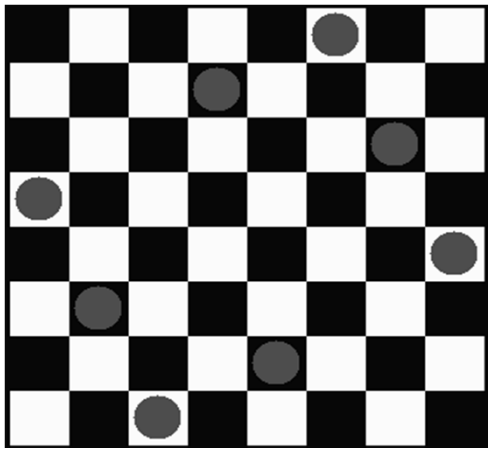
# Criptoaritmetica



```
set of int: Digit = 0..9;

var Digit: S;var Digit: E;var Digit: N;
var Digit: D;var Digit: M;var Digit: O;
var Digit: R;var Digit: Y;

include "all_different.mzn";

constraint all_different([S,E,N,D,M,O,R,Y]);

constraint      1000 * S + 100 * E + 10 * N + D
              + 1000 * M + 100 * O + 10 * R + E
            =  10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint M > 0 /\ S > 0;

solve satisfy;

output ["    ", show(S), show(E), show(N), show(D),"\n",
         " + ", show(M), show(O), show(R), show(E),"\n",
" = ", show(M), show(O), show(N), show(E), show(Y),"\n"];
```

# n-regine

Mettere n regine su una scacchiera nxn in modo che nessuna regina ne attacchi un'altra



Modello CSP

- Variabili: $\{Q_1,Q_2,Q_3,Q_4,\ldots,Q_n\}$.
- Domini: {1,2,3,…,n} rappresenta la colonna in cui sta la variabile.
- Vincoli
  - Regine non nella stessa riga: implicito nelle variabili.
  - Regine non nella stessa colonna: $Q_i$ != $Q_j$
  - Regine non nella stessa diagonale: $|Q_i-Q_j|$ != $|i-j|$

# n-regine



```
include "globals.mzn";

int: n=8;
array[1..n] of var 1..n: queens;

solve satisfy;

constraint all_different(queens) :: domain;
constraint all_different([queens[i]+i | i in 1..n]) :: domain;
constraint all_different([queens[i]-i | i in 1..n]) :: domain;

output [ show(queens) ++ "\n"] ++
[ if j = 1 then "\n" else "" endif ++
     if fix(queens[i]) = j then
        show_int(2,j)
     else
        "__"
     endif
  | i in 1..n, j in 1..n
] ++
["\n"];
```

# Sequenza magica



Trovare una lista di numeri $S$= [$s_0$,…,$s_{n-1}$] tale che $s_i$ sia il numero di volte in cui il numero $i$ appare in $S$.

Esempi: (2, 0, 2, 0) oppure (1,2,1,0)

Modello CSP

Variabili: $s_0$,$s_1$, …,$s_{n-1}$

Domini:{0,1,…,n}

Vincoli:

- Numero di occorrenze di i in ($s_0$,$s_1$, …,$s_{n-1}$) [ $s_i$.
- Vincoli ridondanti:
  - La somma di $s_0$,$s_1$, …,$s_{n-1}$ è n
  - La somma di i*$S_i$, i in [0..n-1], è n

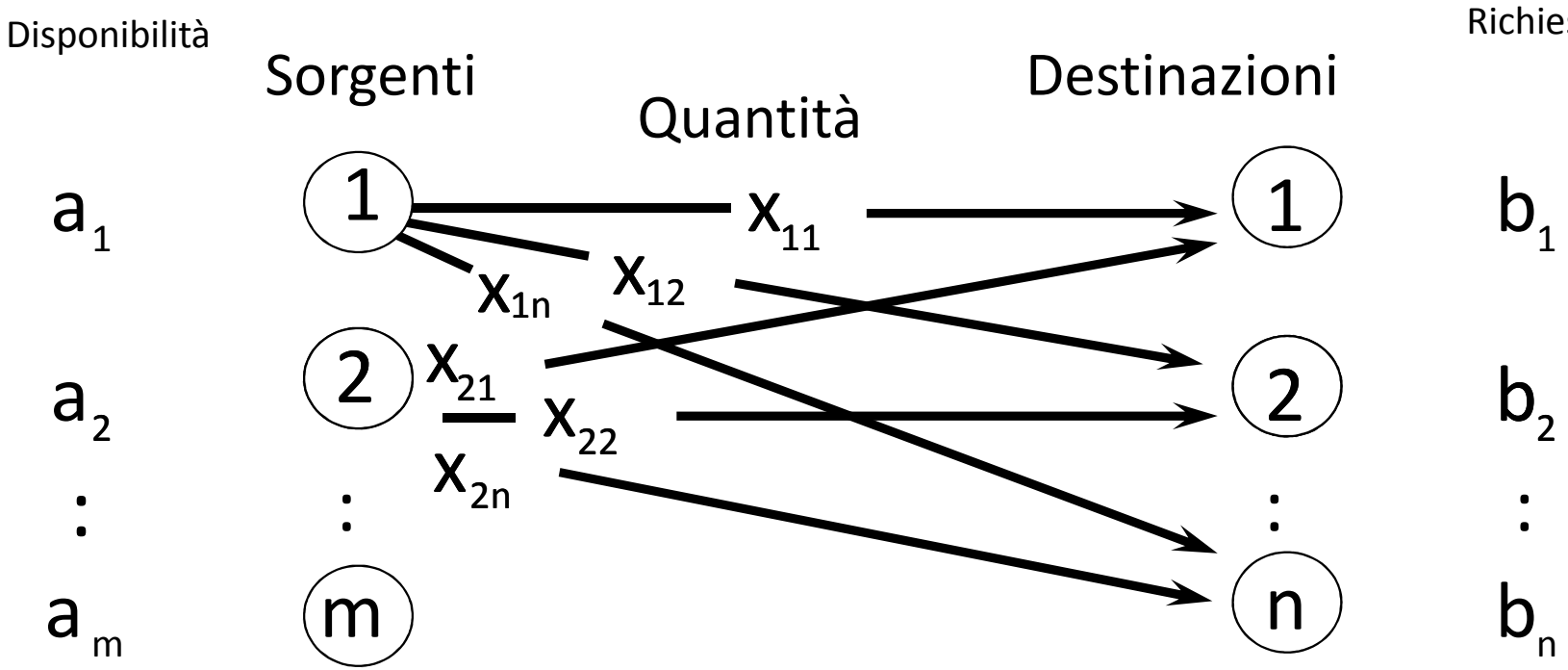
# Sequenza magica



```
int: n =  20;
array[0..n-1] of var 0..n-1: s;

solve satisfy;

constraint
  forall(i in 0..n-1) (
    s[i] = sum(j in 0..n-1) (bool2int(s[j] = i))
  )
;

output [ show(s), "\n" ];
```

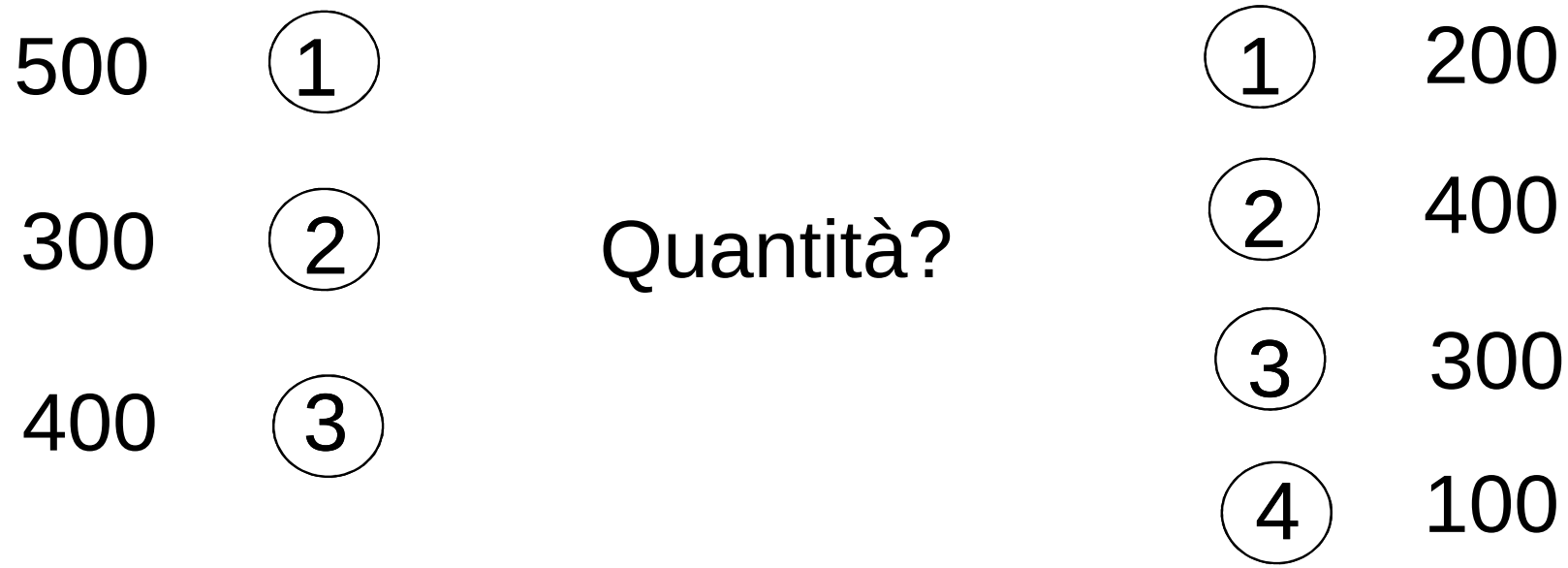
# Problema dei trasporti

Dati un insieme di sorgenti, con relative disponibilità, e uno di destinazioni, con relative richieste, dati i costi di trasporto, determinare il piano di trasporto a costo minimo per soddisfare tutte le richieste.



# Problema dei trasporti

Tiny instance:
3 impianti con capacità note, 4 clienti con domanda nota, costi di trasporto per unità di peso fra ogni coppia.



# Problema dei trasporti



Modello CSP di ottimizzazione (*mzn-g12mip*):

- Variabili: xij, fra ogni coppia OD
- Dominio: integer
- vincoli:
  - Vincoli sulla domanda:

$$\sum_i x_{ij} = b_j \qquad j = 1 \dots n$$

  - vincoli sulla capacità:

$$\sum_j x_{ij} \leq a_i \qquad i = 1 \dots m$$

- Funzione obiettivo (min): $\sum_{ij} c_{ij} x_{ij}$

# Problema dei trasporti

```
int: num_plants;
int: num_clients;

array[1..num_plants] of int: capacity;
array[1..num_clients] of int: demand;
array[1..num_plants, 1..num_clients] of int: cost;
array[1..num_plants, 1..num_clients] of var int: x;

var int: z = sum(i in 1..num_plants, j in 1..num_clients)
(x[i,j]*cost[i,j]);

solve minimize z;

constraint
        forall(i in 1..num_plants, j in 1..num_clients) (
           x[i,j] >= 0
        )
        /\
        forall(i in 1..num_plants) (
           sum(j in 1..num_clients) (x[i,j]) <= capacity[i]
        )
        /\
        forall(j in 1..num_clients) (
           sum(i in 1..num_plants) (x[i,j]) = demand[j]
        );
```

# Di chi è la zebra?



Cinque uomini di 5 diverse nazionalità vivono nelle prime 5 case di una strada. Esercitano 5 diverse professioni e ognuno possiede un animale e ha una bevanda e una marca di tabacco (probl. del 1962) preferita, tutti diversi. Le 5 case sono dipinte di 5 colori diversi. Il problema è determinare chi beve acqua e chi possiede la zebra.

v. http://www.emn.fr/z-info/sdemasse/gccat/Kzebra_puzzle.html

Modello

- Variabili: persone, colori, animali, bevande, tabacchi.
- Domini: 1..5
- Vincoli:

  I 15 elencati nel puzzle.

***Esercizio!***

# Knapsack

E' dato un knapsack (uno zaino) di capacità limitata e un insieme di elementi, per ciascuno dei quali si specifica peso e valore.

Il problema richiede di determinare il sottinsieme di elementi di valore massimo che può essere contenuto nello zaino. E' un classico problema MIP.

Modello CSP:

- Variabili: per ogni elemento, flag (*take*) che dice se lo si prende o no.
- Dominio: int
- Vincoli: somma dei pesi oggetti scelti minore della capacità
- Funzione obiettivo (max): somma dei valori degli oggetti nel knapsack

# Knapsack

```
int: n;           % number of objects
int: weight_max; % maximum weight allowed (capacity of the
knapsack)
array[1..n] of int: values;
array[1..n] of int: weights;
array[1..n] of var int: take; % 1 if we take item i; 0
otherwise

var int: profit = sum(i in 1..n) (take[i] * values[i]);
solve maximize profit;

constraint        % all elements in take must be >= 0
forall(i in 1..n) ( take[i] >= 0 )   /\
sum(i in index_set(weights))( weights[i] * take[i] ) <=
weight_max;

output [show(take), "\n"];
```