OPGAVE 1: FLOOR PUZZLE

Loes, Marja, Niels, Erik en Joep wonen in een gebouw met 5 verdiepingen, elk op een eigen verdieping. Loes woont niet in de bovenste verdieping. Marja woont niet op de begane grond. Niels woont niet op de begane grond en ook niet op de bovenste verdieping. Erik woont (tenminste één verdieping) hoger dan Marja. Joep woont niet op een verdieping één hoger of lager dan Niels. Niels woont niet op een verdieping één hoger of lager dan Marja. Waar woont iedereen?

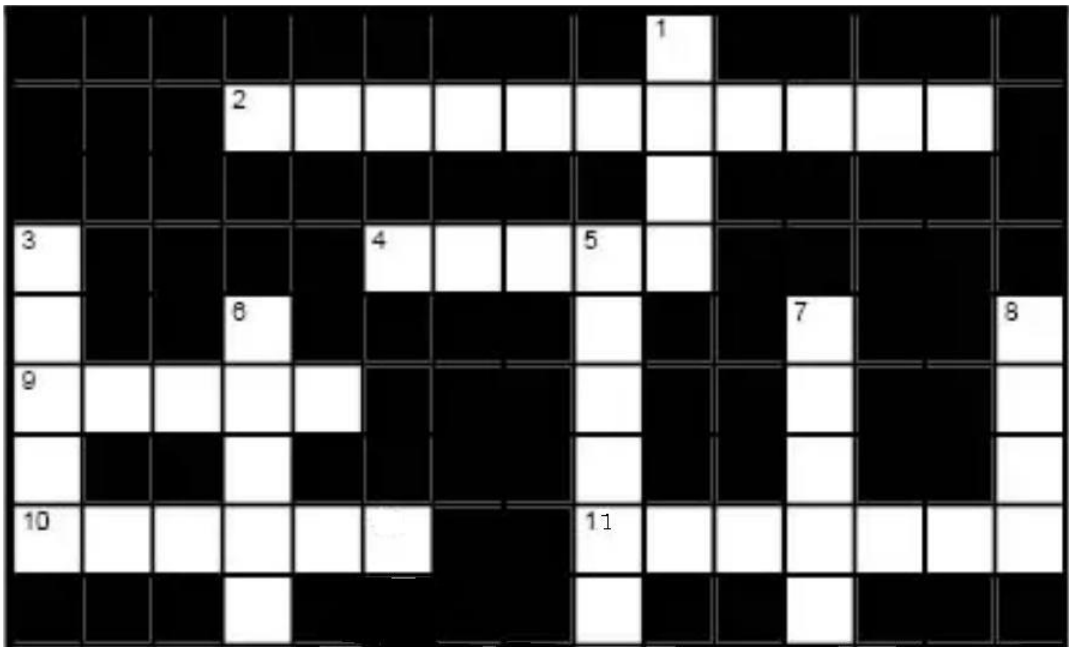
Je kan alle permutaties generen en ze één voor één testen. Dit kan met:

```
for (L, M, N, E, J) in list(itertools.permutations(floors))
```

Voor de performance maakt het uiteraard wel uit in welke volgorde je de testen doet.

OPGAVE 2: KRUISWOORD PUZZEL

Hieronder is een rooster voor een kruiswoordpuzzel afgebeeld. De bedoeling is dat het rooster wordt gevuld met woorden om een kruiswoordpuzzel te genereren. Gegeven is ook een lijst met woorden die gebruikt kunnen worden. Deze lijst words_NL.txt is te vinden op Blackboard. Dit probleem kunnen we zien als een CSP.



1. Vragen m.b.t. variabelen:
    a. Wat zijn hier de variabelen en hoeveel zijn er?
    b. Wat zijn de domeinen van de variabelen?
    c. Hoe kun je de variabelen representeren in code?
    d. Wat is de beste volgorde (prioriteit) voor het testen van variabelen?
2. Vragen m.b.t. constraints:
    a. Wat zijn de binaire constraints en hoeveel zijn er?
    b. Hoe kun je de binaire constraint representeren in code?
    c. Wat betekent het dat een waarde voor een variabele 'arc-consistent' is?
    d. Hoe zinvol is het om arc-consistency toe te passen?

3. Is het mogelijk om alle oplossingen te vinden?

Schijf een programma dat gegeven de lijst met woorden het rooster vult met passende woorden zodat aan alle constraints wordt voldaan. Het is handig de lijst met woorden eerst te verwerken naar datastructuren die de verschillende domeinen representeren. Het programma hoeft alleen maar woorden voor dit specifieke rooster te genereren, dus een generieke opzet wordt niet gevraagd. Er zijn heel veel oplossingen, maar <u>één laten zien</u> is voldoende.

Op Blackboard is de file start_x_word_puzzle.py te vinden die wat suggesties geeft hoe je het kan aanpakken. Merk op dat je het kan zien als één rooster, maar ook als twee onafhankelijke roosters.

OPGAVE 3: KAARTEN PUZZEL

Een kaarten puzzel is als volgt. Er zijn acht kaarten: twee Azen, twee Heren, twee Dames en twee Boeren. De acht kaarten moeten worden geplaatst op het bord (rooster) zoals onder afgebeeld, zodanig dat:

1. elke Aas grenst aan een Heer
2. elke Heer grenst aan een Vrouw
3. elke Vrouw grenst aan een Boer
4. een (elke) Aas grenst NIET aan een Vrouw
5. twee kaarten van dezelfde soort mogen geen buren zijn

		0	
1	2	3	
	4	5	6
		7	

Grenzen aan betekent hier: horizontaal of verticaal.

a) Laten we alle posities op het bord een index geven, en het bord representeren door een dictionary (key=index, value=kaart). Maak een programma die beide oplossingen van de puzzel terug geeft op basis van brute force, dus alle **permutaties** generen en ze één voor één testen. Op Blackboard is de file start_card_puzzle_dfs.py te vinden waarmee je kan beginnen.
    1. Hoeveel (verschillende) permutaties zijn er eigenlijk? Hoe kun je dit zelf berekenen? Hou er rekening mee dat (A,A,A,A …) niet kan, je kan immers A maar 2 keer kiezen.
    2. Bij het testen is het slim om de test die het meest beperkend is het eerst te doen (‘fail fast’). Hoeveel permutaties moeten er worden getest (= iteraties) om de eerste en tweede oplossing te vinden?

b) Implementeer een betere versie door gebruik te maken van DFS en backtracking. Dit kan in ca. 70 regels, waarbij misschien het lastigste is de testen correct te implementeren. Hoeveel iteraties (recursive calls) zijn nu nodig om de eerste en de tweede oplossing te vinden?

    Tip: De testen bij (a) kan je hergebruiken bij (b), alleen bij (b) moet je er rekening mee houden dat een vakje ook leeg kan zijn die pas later (dieper in de DFS-boom) wordt gevuld. De permutaties bij (a) zijn de leaf-nodes van de DFS-boom bij (b).

c) Je kan de puzzel ook oplossen door herhaaldelijk arc-consistency toe te passen (en wat logisch redeneren). Doe dit op papier. Beredeneer dat bord[5] een Heer moet zijn, door te laten zien dat bord[5] geen Aas, Vrouw of Boer kan zijn. Beredeneer daarna - op eenzelfde manier - dat bord[0] ook een Heer moet zijn.

Een (start) voorbeeld hoe je dat kan opschrijven is als volgt.

```
Stel 5 is een Aas.
```

- `3,4,6,7 kunnen geen A zijn vanwege [5]`
- `3,4,6,7 kunnen geen V zijn vanwege [4]`
- `dus 3,4,6,7 moet een H of B zijn`
- `er zijn maar 2xH en 2xB kaarten, dus 0,1,2 moet een A of V zijn`

		A,H,V,B	
A,H,V,B	A,H,V,B	A,H,V,B	
	A,H,V,B	A	A,H,V,B
		A,H,V,B	

OPGAVE 4: SUDOKU MET BACKTRACKING EN ARC-CONSISTENCY

In deze opgave gaan we een 3x3 Sudoku oplossen. Een beschrijving van deze puzzel is op Wikipedia te vinden: link. Zie ook de college sheets.

Eerst wat terminologie:

- er zijn 9x9 cellen, 9 rijen, 9 kolommen en 9 dozen (boxes);
- gelijken (peers) zijn cellen in dezelfde rij, kolom of doos;
- het domein van een variabele zijn alle mogelijke waarde, dus 1..9.

a) Op blackbord is een file sudoku_start.py te vinden die nog moet worden afgemaakt. Maak deze versie af door eerst backtracking (DFS) te implementeren. Let hierbij op de volgorde van variabelen.

b) Verbeter de versie uit (a) door arc-consistency toe te passen. Dit gaat als volgt:

- geef een variabele een waarde;
- pas “arc-consistency” toe (verwijder waarden uit domeinen);
- als een domein leeg wordt, dan gaan we niet verder in de boom (backtracking);
- we proberen een andere waarde toe te wijzen.

De oplossing kan in ongeveer 90 regels.

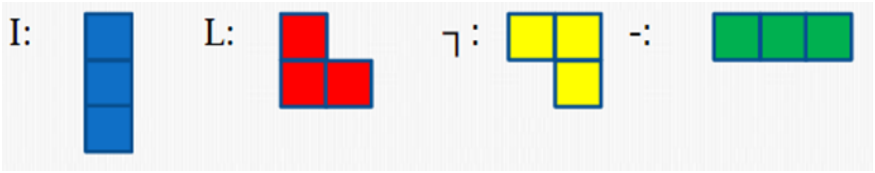
Een lijst met lastige puzzels is hier te vinden: top95. En hier kan je ze zelf genereren: qqwing.

(*Opmerking*: het oplossen van Sudoku #18 uit sudoku_start.py duurde bij mij bijna 2 seconden, de overige (veel) minder dan 1 seconde).

OPGAVE 5: TRIOMINO EN SUDOKU MET ALGORITME-X

In de vorige opgave zagen we dat Sudoku kan worden opgelost op basis van backtracking en arc-consistency. Een andere strategie is het te beschouwen als een exact cover probleem. Een exact cover probleem kan worden opgelost met “Algoritme X”. Om beter te begrijpen hoe dit algoritme werkt gaan we in (a) eerst een eenvoudige puzzel oplossen met Algoritme X, en pas in (b) gaan we dit toepassen op Sudoku.

a) Zoals besproken in het college kan je met Algoritme X pentomino’s en andere ‘tegel’ puzzels oplossen (zie wiki en youtube met Alex Bellos). Een eenvoudige tegelpuzzel is triomino. Er zijn hierbij 4 mogelijke vormen:



Voor de eenvoud nemen we aan dat we ze niet mogen roteren of omdraaien. We plaatsen deze triomino’s op een 3 x 4 bord. Een voorbeeld van een oplossing is dan:

Hoeveel oplossingen zijn er?

Op Blackboard is de file start_triomino.py te vinden. Implementeer hierin Algoritme X zodat alle oplossingen gevonden worden. Hiervoor moet je nog ongeveer 30 regels toevoegen (alleen de functies cover en solve implementeren).

b) *Deze opgave is niet verplicht, maar na opgave (a) is dit niet heel lastig. Alg-X blijft immers hetzelfde.* Op blackbord is een versie sudoku_start.py te vinden die nog moet worden afgemaakt. Los de Sudoku’s op door Algoritme X zoals gemaakt bij (a) te her-gebruiken. Dit kan in ongeveer 110 regels.