Algorismes de força bruta

Algorísmica

Grau d'Enginyeria Informàtica

Mireia Ribera, ribera@ub.edu
Daniel Ortiz, daniel.ortiz@ub.edu

Algorismes de força bruta: Conceptes clau

1. Definició
2. Concepte d’intractabilitat, cas fort i cas dèbil
3. Problemes d’exemple: Ordenació
4. Problemes d’exemple: Motxilla i N Reines
5. Combinacions, permutacions, i algorisme de Johnson-Trotter

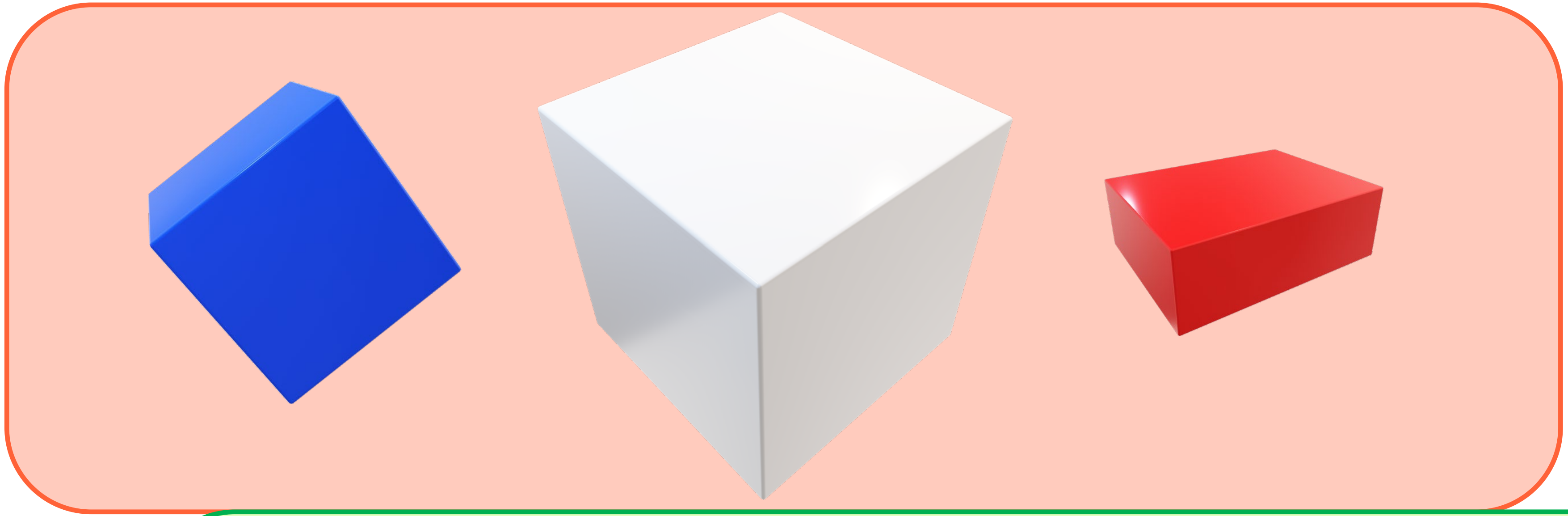
Quins són els algorismes de força bruta (“brute force”)?

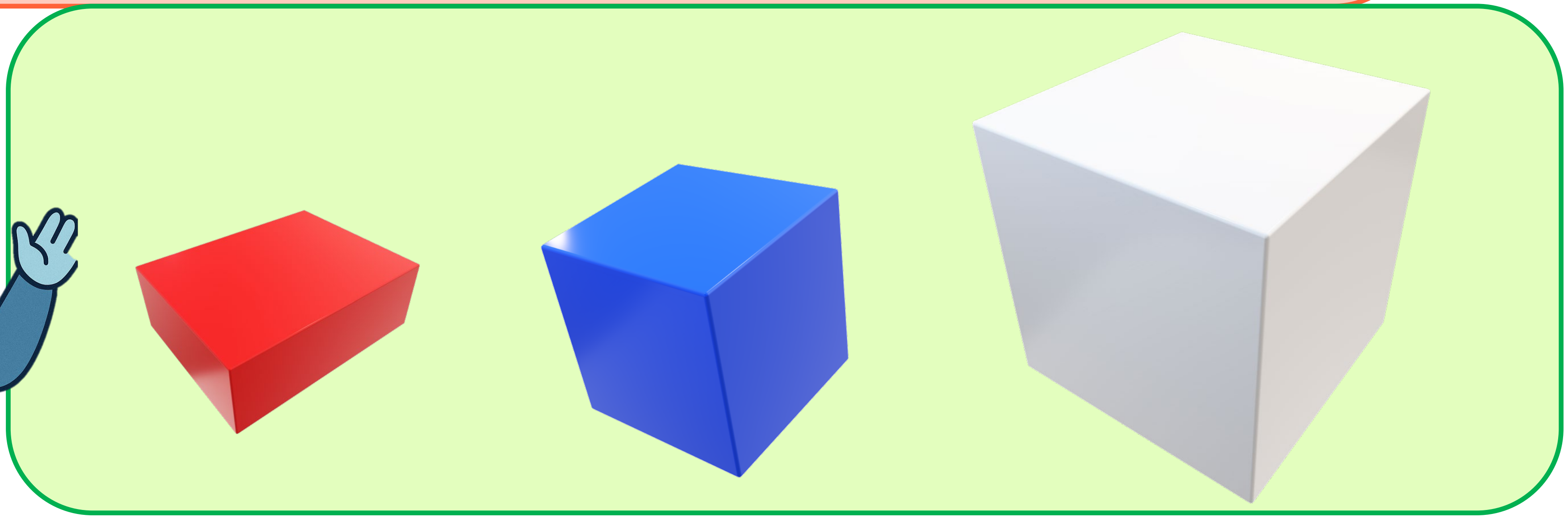
Diem que un algorisme està basat en la força bruta si implementa la solució a un problema basant-se en la definició del problema sense cap mena d’optimització.

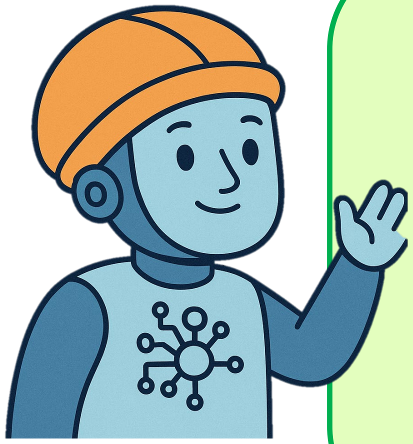
Per exemple quan calculem l’exponent modular, l’aproximació per força bruta seria fer primer l’exponent i després el mòdul.
Hem vist ja que hi havia maneres de calcular-ho molt més òptimes.

Veurem alguns problemes i la seva resolució per força bruta. En alguns casos hi haurà algorismes que els optimitzen i en altres no.

El problema de l’ordenació







Ordenació

Ordenar vol dir posar els elements de menor a major en una seqüència segons un determinat criteri, per exemple:

- Ordenar alumnes per DNI
- Ordenar les factures per import
- Ordenar registres d'una base de dades per data

És una de les operacions més repetides a l'ordinador, i un pas previ a moltes altres operacions.

Més endavant aprendrem algoritmes òptims d'ordenació, en aquest tema en veurem un de força bruta.

Ordenació per selecció

- Tenim una llista A d'n elements.

  Per ex. $A = [23{,}27{,}5{,}90{,}12{,}44{,}38{,}84{,}77]$
- Recorrem la llista per trobar l'element més petit i el canviem pel primer element.
- Comencem pel segon element i fem el mateix amb els elements que queden a la dreta
- En el pas i $(0 \leq i \leq n-2)$, busquem l'element més petit a $A[i+1..n-1]$, i el canviem per $A[i]$

0	1	2	3	4	5	6	7	8
23	17	5	90	12	44	38	84	77

0	1	2	3	4	5	6	7	8
5	17	23	90	12	44	38	84	77

5	12	23	90	17	44	38	84	77

5	12	17	90	23	44	38	84	77

•••

5	12	17	23	38	44	77	84	90

5	12	17	23	38	44	77	84	90

Ordenació per selecció

```python
def ordenacio_seleccio(llista:list[int])->None:
    for i in range(0, len(llista)-1):
        min:int = i
        for j in range(i + 1, len(llista)):
            if llista[j] < llista[min]:
                min = j
        llista[i],llista[min]=llista[min],llista[i]
```

Hi ha dues iteracions imbricades i a dins una comparació i una assignació. Quantes vegades s’executen aquestes operacions interiors?

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \left(\sum_{i=1}^{n} i\right) - n = \frac{n(n+1)}{2} - n \approx O(n^2)$$

Ordenació per selecció

Atenció!

🚨 No useu mai un algorisme d'ordre quadràtic $O(n^2)$ per ordenar.

Més endavant veurem que hi ha algorismes de complexitat $O(n \log n)$

Concepte d’intractabilitat, cas fort i cas dèbil

Per a alguns problemes no s’ha trobat cap solució eficient i tenen ordres exponencials o factorials i per tant són inviables per a n grans.

En veurem dos, el problema de la motxilla $O(2^n)$ i el problema de les N Reines O(n!).

Fins ara havíem vist problemes de tipus **P** (polinomials), els problemes de les N Reines o de la motxilla són problemes que anomenem **intractables.**

- **Cas fort:** S’ha demostrat que no existeix un algorisme per resoldre el problema
- **Cas dèbil:** No es coneix cap algorisme eficient.

Quan no hi ha cap algorisme eficient per resoldre el problema, sovint ens caldrà 1) enumerar totes les solucions i 2) decidir la millor.

🛠️ Combinacions, permutacions i producte cartesià

Molts problemes de força bruta o cerca exhaustiva requereixen generar tots els possibles arranjaments d'un conjunt d'elements. Això s'estudia a combinatòria.

Vegem els principals arranjaments:

- ❑ Permutacions
- ❑ Combinacions
- ❑ Producte cartesià

A Python les funcions relacionades amb la combinatòria a Python estan dins la llibreria itertools.

🛠️ Permutacions

Permutació de k elements d'un conjunt donat de n elements (conjunt, num_elements): són tots els possibles arranjaments d'aquell nombre d'elements, l'ordre importa.

Exemple: Si el conjunt és (A,B,C), les permutacions de dos elements són (A,B), (A,C), (B,A), (B,C), (C,A), (C,B)

Hi ha tantes permutacions com $n!/(n-k)!$

```
from itertools import permutations

elements = ["A", "B", "C", "D"]
print("Permutacions de dos elements")
for p in permutations(elements,2):
    print(p,end=", ")

> Permutacions de dos elements
> ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'C'), ('B', 'D'), ('C', 'A'), ('C', 'B'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D', 'C')
```

Combinacions

Combinació de k elements d'un conjunt donat de n elements (conjunt, num_elements): són tots els possibles arranjaments, sense importar l'ordre.

Exemple: Si el conjunt és (A,B,C), les combinacions de dos elements són {A,B} {A,C}, {B,C}

Hi ha tantes combinacions com $n!/k!\,(n-k)!$

```
from itertools import combinations
elements = ["A", "B", "C", "D"]

print("Combinacions de tres elements")
for c in combinations(elements,3):
    print(c, end=", ")

➢ Combinacions de tres elements
➢ ('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'C', 'D'), ('B', 'C', 'D'),
```

Producte cartesià

Producte cartesià (tots els elements): Fa arranjaments entre els elements de dos conjunts. A Python també permet definir-lo dins un conjunt amb repetició d'elements.

Exemple: Si el conjunt_A és [1,2] i el conjunt_B és ["vermell", "blau", "verd"], el producte cartesià AxB és (1, 'vermell'), (1, 'blau'), (1, 'verd'), (2, 'vermell'), (2, 'blau'), (2, 'verd')
Exemple: Si el conjunt és (A,B), el producte cartesià amb repetició és (X,X), (X,Y), (Y,X), (Y,Y)

Hi ha tants productes cartesians com $|A|x|B|$

```
from itertools import product
conjunt_A = ["A", "B"]
conjunt_B = [1, 2, 3]

for prod in product(conjunt_A, conjunt_B):
    print(prod,end=", ")
```

➢ ('A', 1), ('A', 2), ('A', 3), ('B', 1), ('B', 2), ('B', 3),

Notebook de suport

Consulta tot el codi del tema al notebook: Tema5ForcaBruta-Codi.ipynb

Permutacions: algorisme de Steinhaus-Johnson-Trotter

Aquest algorisme ens permet crear permutacions d’una manera molt eficient.

Hugo Steinhaus

Font: Wikipedia

Wladyslaw Hugo Dionizy Steinhaus, era un matemàtic polac i va publicar un llibre l’any 1958 on apareix un trencaclosques relacionat amb les permutacions.

Selmer Martin Johnson va ser un matemàtic americà (de la corporació RAND) i Hale Freeman Trotter va ser un matemàtic canadenc-americà. Ambdós van descobrir l’algorisme el 1960 cadascun per la seva banda.

Font: alchertron.com

Selmer M. Johnson

Font: Wikipedia

Hale Trotter

Permutacions: algorisme de Steinhaus-Johnson-Trotter

<1	<2	<3	<4
<1	<2	<4	<3
<1	<4	<2	<3
<4	<1	<2	<3
4>	<1	<3	<2
<1	4>	<3	<2
<1	<3	4>	<2
<1	<3	<2	4>
<3	<1	<2	<4
<3	<1	<4	<2

<3	<4	<1	<2
<4	<3	<1	<2
4>	3>	<2	<1
3>	4>	<2	<1
3>	<2	4>	<1
3>	<2	<1	4>
<2	3>	<1	<4
<2	3>	<4	<1
<2	<4	3>	<1
<4	<2	3>	<1

4>	<2	<1	3>
<2	4>	<1	3>
<2	<1	4>	3>
<2	<1	3>	4>

Permutacions: algorisme de Steinhaus-Johnson-Trotter

```
while hi_ha_moviment:

    element_mobil:int = -1
    index_mobil:int = -1
    trobat_mobil:bool = False

    for i in range(n):
        k = p[i]
        adj_index:int = i - 1 if direccions[i] == False else i + 1
        if 0 <= adj_index < n and k > p[adj_index] and k > element_mobil:
            element_mobil = k
            index_mobil = i
            trobat_mobil = True
    hi_ha_moviment = trobat_mobil

    target_index:int = index_mobil - 1 if direccions[index_mobil] == False
                                          else index_mobil + 1
    p[index_mobil], p[target_index] = p[target_index], p[index_mobil]
    direccions[index_mobil], direccions[target_index] =
              direccions[target_index], direccions[index_mobil]

    for i in range(n):
        if p[i] > element_mobil:
            direccions[i] = not direccions[i]
```

Problemes intractables, d’ordre exponencial o factorial

Veurem dos problemes intractables:

- ❑ El problema de la motxilla
- ❑ El problema de les N Reines

Recordeu el problema de la motxilla?

Ara que sabem programar i que sabem calcular complexitats, veurem que **és un problema O(2^n)!**

Repassem la definició del problema

La motxilla pot portar un pes màxim de 2.3kg

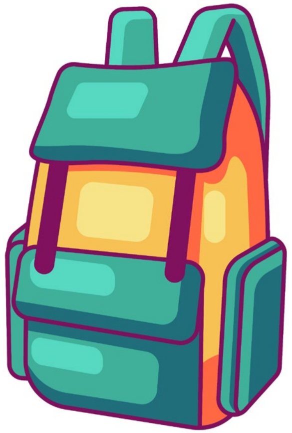
Disposem d'un seguit d'objectes que hi volem posar, cadascun amb un pes $wobj_i$ i amb un valor $vobj_i$. Exemple: la bufanda pesa 0.2kg (wobj) i té un valor de 30€ (vobj).

Volem triar aquells elements que maximitzin el valor de la motxilla un cop carregada.

Aquest és un problema genèric per a portafolis financers, càrregues de vaixells o avions, tall de materials o emplenat de contenidors.

Quina és l'*entrada*? Quina la *sortida esperada*? Com es defineix *correcte*? i *eficient*?

I com arribàvem a la solució:



2.3kg

Botes 2kg 150€	Mòbil 0.4kg 300€	Bufanda 0.2Kg 30€	Crema 0.5Kg 50€	Pes	Valor
0	0	0	1	0.5kg	50
0	0	1	0	0.2kg	30
0	0	1	1	0.7kg	80
0	1	0	0	0.4kg	300
0	1	0	1	0.9kg	350
0	1	1	0	0.6kg	330
0	1	1	1	1.1kg	380
1	0	0	0	2kg	150
~~1~~	~~0~~	~~0~~	~~1~~	~~2.5kg~~	~~200~~
1	0	1	0	2.2kg	180
~~1~~	~~0~~	~~1~~	~~1~~	~~2.7kg~~	~~230~~
~~1~~	~~1~~	~~0~~	~~0~~	~~2.4kg~~	~~450~~
~~1~~	~~1~~	~~0~~	~~1~~	~~2.9kg~~	~~500~~
~~1~~	~~1~~	~~1~~	~~0~~	~~2.6kg~~	~~480~~
~~1~~	~~1~~	~~1~~	~~1~~	~~3.1kg~~	~~530~~

Motxilla: codi

```python
from itertools import combinations
def motxilla_fb(nombre, capacitat, pes_cost):
    """
    :param nombre: nombre d'items possibles
    :param capacitat: capacitat de la motxilla
    :param pes_cost: llista de tuples del tipus [(pes, cost), (pes, cost), ...]
    """
    millor_cost = None
    millor_combinacio = []
    # genera totes les combinacions possibles d'1 o més elements
    for num_elements in range(nombre):
        for comb in combinations(pes_cost, num_elements + 1):
            pes = sum([pc[0] for pc in comb])
            cost = sum([pc[1] for pc in comb])
            if (millor_cost is None or millor_cost < cost) and pes <= capacitat:
                millor_cost = cost
                millor_combinacio = [0] * nombre
                for pc in comb:
                    millor_combinacio[pes_cost.index(pc)] = 1
    return millor_cost, millor_combinacio
```

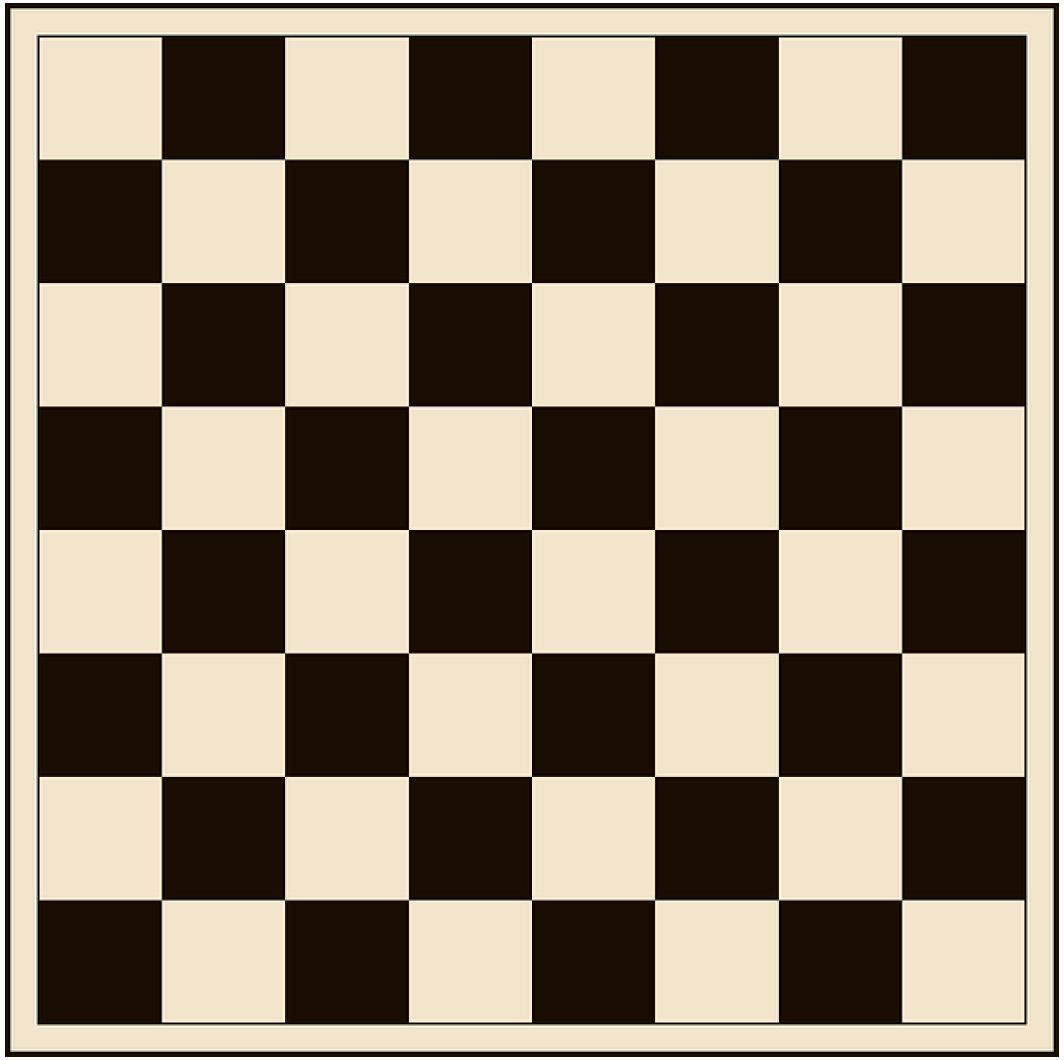
Motxilla: Càlcul del cost

Podem assimilar la decisió d'incloure un element o no a la motxilla a un dígit binari: un 1 vol dir que l'incloem, un 0 que no l'incloem.

Si tenim n elements tenim 2^n possibilitats. Les hem de crear, comptar el pes i el valor i decidir quina és la de més valor sense sobrepassar el pes màxim.

L'ordre de complexitat és 2^n , exponencial!

El problema de les N Reines



Definició del problema

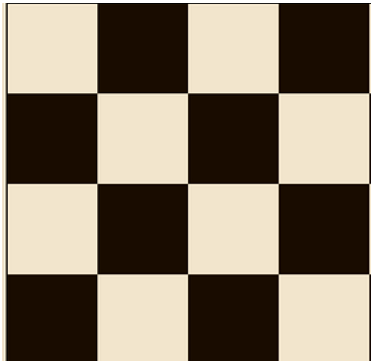
🎯 Objectiu : el problema de les N Reines consisteix a col·locar N peces de reina en un tauler d'escacs de $N \times N$ caselles de tal manera que cap reina pugui atacar-ne cap altra.

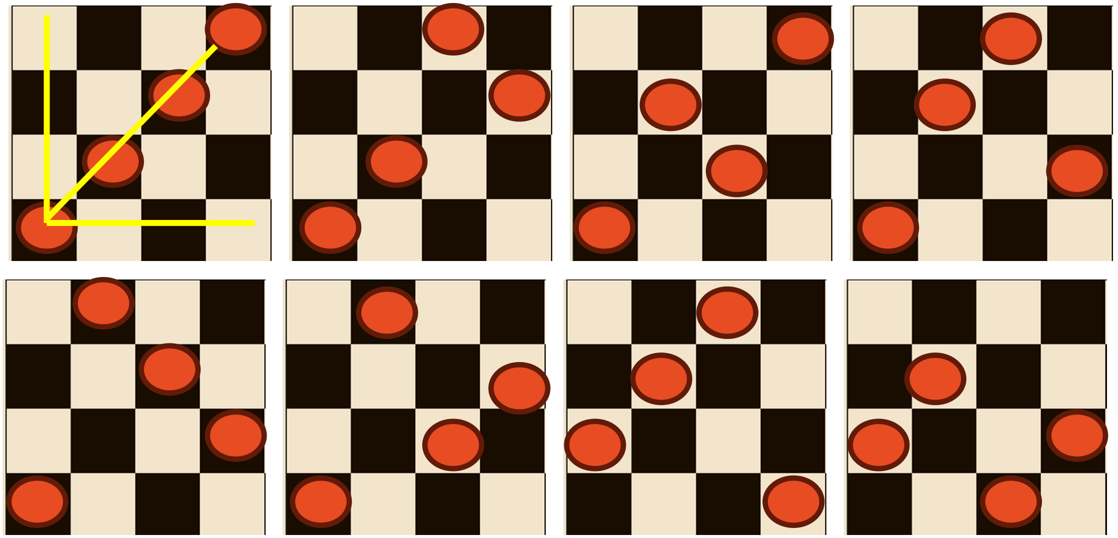
👑 Regles dels Escacs Aplicades

En el context d'aquest problema, dues reines s'ataquen si es troben:

- ❑ A la mateixa fila horitzontal.
- ❑ A la mateixa columna vertical.
- ❑ A la mateixa diagonal (ascendent o descendent).

El problema de les N Reines

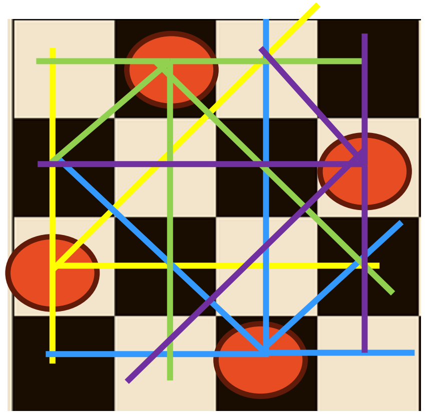




... 4 x 3 x 2 x 1 permutacions, 4!

♟ Algorisme: per trobar la solució farem totes les permutacions d'una reina per columna (0,1,2,3)(0,1,3,2) (0,2,1,3)...

I després validarem que no es puguin matar entre elles.



Amb N=4 hi ha 2 solucions:
(1, 3, 0, 2) (2, 0, 3, 1)

Aquesta és una possible solució: a la columna 0 la reina és a la posició 1, a la columna 1, la reina és a la posició 3...
(1,3,0,2)

Reines: codi

```python
def hi_ha_atac_diagonal(reines_per_columna:list[any])->bool:
    N = len(reines_per_columna)

    for i in range(N):
        for j in range(i + 1, N):
            if abs(reines_per_columna[i] - reines_per_columna[j]) == abs(i - j):
                return True
    return False

def n_reines_forca_bruta(N:int)->int:
    files:list[int] = range(N)
    totes_disposicions:list(any) = permutations(files)
    solucions:int = 0
    for disposicio_files in totes_disposicions:
        # Si NO hi ha atac diagonal, és una solució vàlida
        if not hi_ha_atac_diagonal(disposicio_files):
            solucions += 1
            print(f"Solució trobada: {disposicio_files}")

    return solucions
```

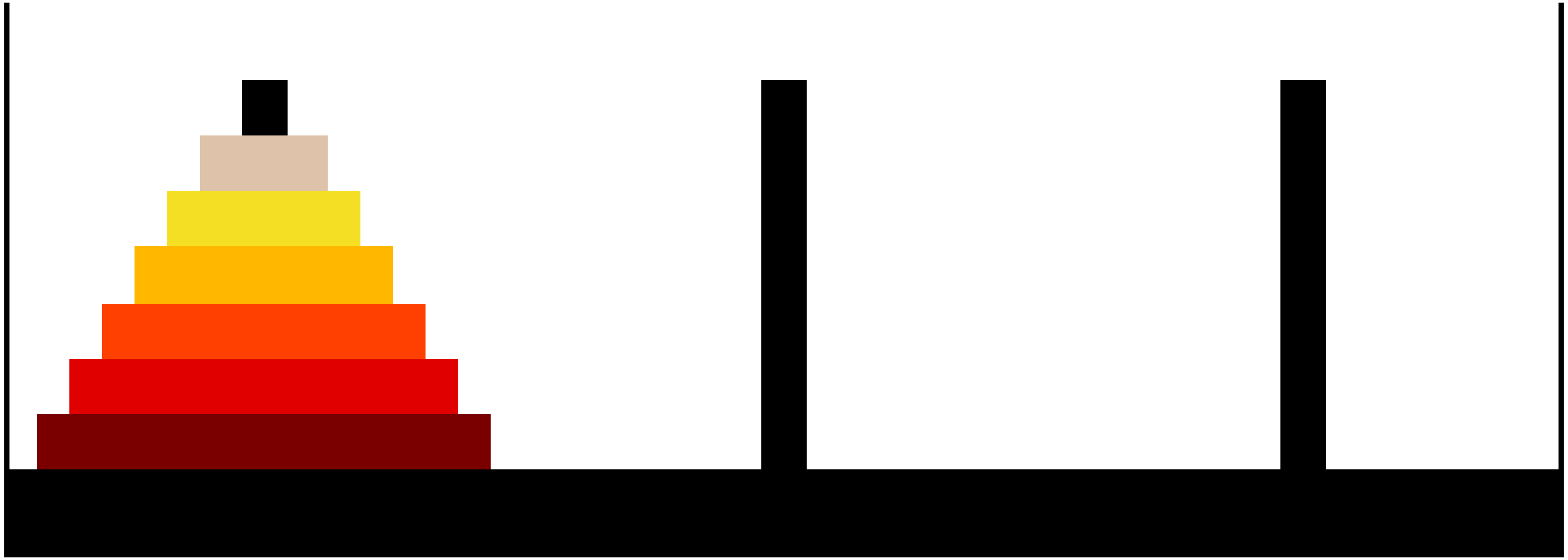
Reines: Càlcul del cost

Hem de crear totes les permutacions possibles.

Si tenim n caselles tenim $n!$ possibilitats. I per cadascuna haurem de vigilar que no mati a una reina d’una altre columna.

L’ordre de complexitat és $n!$, .es a dir, és un ordre factorial!

El problema de les Torres de Hanoi



Aquest és un problema de força bruta, recursiu, i per tant l'hem inclòs en el tema de recursivitat.

Ara et toca...

Una màquina de begudes només accepta monedes de 10 cèntims, 20 cèntims i 50 cèntims. I només accepta un nombre Q de monedes. El preu de la beguda és de P cèntims.
L'objectiu és trobar totes les combinacions exactes de monedes de 10c, 20c i 50c que sumen el preu P.

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

LIBERTAS PERFVNDET · OMNIA LVCE

UNIVERSITAT DE BARCELONA

Recapitulació

🏋️‍♂️🏋️‍♀️ Els algoritmes de força bruta no optimitzen el procés, es basen en la definició bàsica

👌 Alguns problemes tenen solucions molt més òptimes. Per exemple, l’ordenació.

🚨 Hi ha alguns problemes que són intractables

♾️ Sovint cal crear totes les solucions possibles, i triar la que compleix certes restriccions

4> Per a fer-ho cal crear permutacions i combinacions. Podem usar Steinhaus-Johnson-Trotter

⏱️ Tenen complexitats factorials o exponencials.

🎒 ♟️ 🍡 Hem vist tres problemes intractables: motxilla, N reines i Hanoi