UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques

Algorísmica

Algorismes i força bruta

Jordi Vitrià

Força bruta...

Diem que un algorisme està basat en la **força bruta** si implementa la solució a un problema basant-se directament en la definició del problema i en la definició dels conceptes involucrats.

- Calcular a^n *mod m* (a>0, n>=0).
- Calcular *n!*
- Multiplicar dues matrius (A·B).
- Buscar el valor mínim en els valors d'una funció.

Ordenació d’una llista

Ordenar és una de les operacions més repetides per qualsevol ordinador!

- Ordenar una llista de persones.
- Ordenar els registres d’una base de dades per data.
- Ordenar les factures per import.
- Ordenar pàgines web a un cercador.
- Ordenar productes en un recomanador.
- Etc.

És més, ordenar és un pas previ per moltes altres operacions computacionals!

Ordenació d'una llista

Hi ha molts algorismes d'ordenació. Anem a veure'n dos basats en la **força bruta**.

Name	Average	Worst	Method
Bubble sort	$O(n^2)$	$O(n^2)$	Exchanging
Cocktail sort	—	$O(n^2)$	Exchanging
Comb sort	—	—	Exchanging
Gnome sort	—	$O(n^2)$	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	Insertion
Shell sort	—	$O(n \log^2 n)$	Insertion
Binary tree sort	$O(n \log n)$	$O(n \log n)$	Insertion
Library sort	$O(n \log n)$	$O(n^2)$	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	Merging
In-place merge sort	$O(n \log n)$	$O(n \log n)$	Merging
Heapsort	$O(n \log n)$	$O(n \log n)$	Selection
Smoothsort	—	$O(n \log n)$	Selection
Quicksort	$O(n \log n)$	$O(n^2)$	Partitioning
Introsort	$O(n \log n)$	$O(n \log n)$	Hybrid
Patience sorting	—	$O(n \log n)$	Insertion & Selection
Strand sort	$O(n \log n)$	$O(n^2)$	Selection
Tournament sort	$O(n \log n)$	$O(n \log n)$	Selection

Ordenació per força bruta: Ordenació per selecció.

Algorisme d'ordenació per selecció:

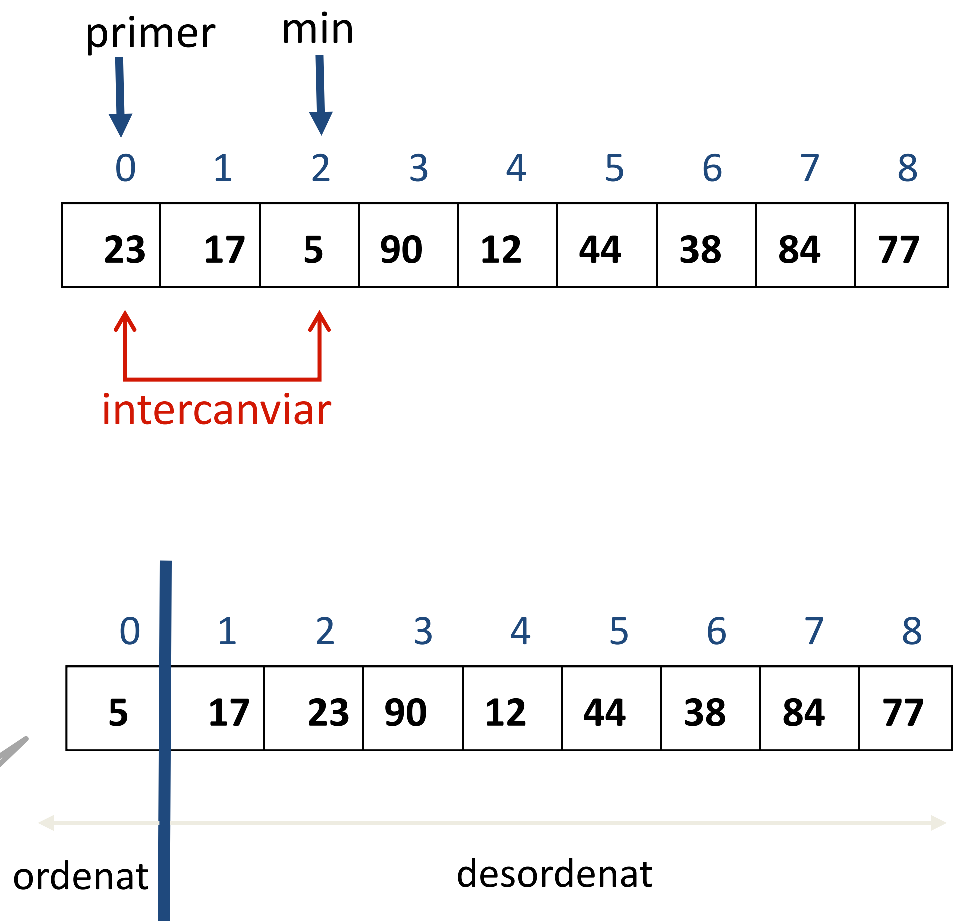
Analogia amb una baralla de cartes

- Recorrem la llista *A* per trobar l'element més petit i el canviem pel primer element.
- Llavors, començant pel segon element, mirem els elements que queden a la dreta i busquem el menor, que canviem pel segon.
- En general, al pas i ($0 \leq i \leq n-2$), busquem l'element més petit a $A[i..n-1]$ i el canviem per $A[i-1]$.

$$A[0] \leq \ldots \leq A[i-1] \mid A[i], \ldots, A[min], \ldots, A[n-1]$$

Ordenació per força bruta: Ordenació per selecció.

1. Trobar l’element més petit de la llista.
2. Intercanviar l’element a la primera posició amb l’element més petit. Ara aquest és a la primera posició.
3. Repetir els passos 1 i 2 amb la llista després de descartar el primer element que ja està ordenat.



Resultat d’una passada.

Ordenació per força bruta: Ordenació per selecció.

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

•••

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

Ordenació per força bruta: Ordenació per selecció.

Algorisme d'ordenació per selecció (Python):

```
def selection_sort(l):
    for i in range(0, len(l)-1):
        min = i
        for j in range(i + 1, len(l)):
            if l[j] < l[min]:
                min = j
        l[i],l[min]=l[min],l[i]
```

```
>>> a=[8,7,6,5,4,3,2]
>>> selection_sort(a)
>>> a
[2, 3, 4, 5, 6, 7, 8]
```

Ordenació per força bruta: Ordenació per selecció.

Complexitat de l'algorisme d'ordenació per selecció:

L'operació més important és una comparació:

```
if l[j] < l[min]: min = j
```

I el nombre de vegades que s'executa és:

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

Evidentment l'algorisme és quadràtic, tot i que només fem *O(n)* intercanvis a la llista.

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

Hi ha un altre estratègia «simple» per ordenar,que simula el **comportament d'una bombolla**...
La idea és comparar elements adjacents a la llista i intercanviar-los si estan mal ordenats: **a la primera passada l'element més gran serà l'últim.**

$$\boldsymbol{A[0] ,. . . , A[i\text{-}1] \overset{?}{\leftrightarrow} A[i], . . . , A[n\text{-}1]}$$

A la passada *i* tindrem els elements de *n-i* fins *n-1* ordenats:

$$A[0] ,. . . , A[j] \leftrightarrow A[j+1], . . . , A[n-i-1] \;|\; A[n-i] \leq . . . \leq A[n-1]$$

Seguint aquesta estratègia, en n-1 passades obtenim una llista ordenada.

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

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

intercanvi

17	**23**	**5**	**90**	**12**	**44**	**38**	**84**	**77**

intercanvi

17	**5**	**23**	**90**	**12**	**44**	**38**	**84**	**77**

ok

intercanvi

17	**5**	**23**	**12**	**90**	**44**	**38**	**84**	**77**

intercanvi

17	**5**	**23**	**12**	**44**	**90**	**38**	**84**	**77**

intercanvi

17	**5**	**23**	**12**	**44**	**38**	**90**	**84**	**77**

intercanvi

17	**5**	**23**	**12**	**44**	**38**	**84**	**90**	**77**

intercanvi

17	**5**	**23**	**12**	**44**	**38**	**84**	**77**	**90**

El valor més gran, 90 és al final de la llista.

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

```
def bubble_sort(l):
    for i in range(len(l)-1):
        for j in range(len(l)-1-i):
            if l[j] > l[j+1]:
                l[j],l[j+1]=l[j+1],l[j]
```

```
>>> a=[8,7,6,5,4,3,2]
>>> bubble_sort(a)
>>> a
[2, 3, 4, 5, 6, 7, 8]
```

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

En el pitjor cas, el bucle exterior s’executa *n*-1 vegades per a fer *n*-1 passades.

Per cada execució del bucle exterior, l’interior s’executa *n*-*i* vegades: El nombre de comparacions a cada passada consecutiva és *n*-1, *n*-2, … , 1.

Sumant tenim el nombre total de comparacions:

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

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

En el pitjor cas, el bucle exterior s'executa n-1 vegades per a fer n-1 passades.

Per cada execució del bucle exterior, l'interior s'executa n-i vegades: El nombre de comparacions a cada passada consecutiva és n-1, n-2, … , 1.

Sumant tenim el nombre total de comparacions:

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

Quin dels dos algorismes d'ordenació fa més intercanvis?

Ordenació per força bruta: Ordenació amb el mètode de la bombolla.

- El comportament de l'ordenació per **bombolla** o per **selecció** **són aproximadament equivalents**, tot i que el de la bombolla fa **molts més intercanvis**!

- Podem fer una petita millora al mètode de la bombolla: si en una passada no intercanviem res, podem acabar!

Algorismes

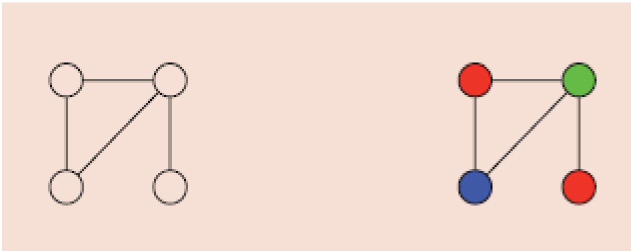
Hi ha molts problemes computacionals que s'han demostrat **intractables**. La intractabilitat pot ser de dos tipus:

- *Cas fort*: S'ha demostrat que no existeix un algorisme per resoldre el problema (p.e. la indecibilitat de l'aturada d'un programa).
- *Cas dèbil*: No es coneix cap algorisme eficient per resoldre el problema (p.e. la factorització).

NOTA: Una forma d'afrontar la intractabilitat són els **algorismes aproximats**.

Cerca exhaustiva

Quan no <u>hi ha cap algorisme eficient</u> per resoldre un problema ens enfrontem a un problema de <u>cerca per força bruta</u>: enumerar totes les solucions i trobar la millor.



Exemple:

- **Coloració d'un graf**: Donat un graf G amb n vèrtexs, m arcs i una paleta de k colors, decidir si és possible assignar a cada vèrtex un color de manera que tots els arcs tenen colors diferents als seus extrems.

Cerca exhaustiva

La **cerca exhaustiva** (o **cerca per força bruta**) consisteix en una exploració sistemàtica de **l'espai de solucions possibles** a un problema donat.

Pot dividir-se en varies parts: **com generar totes les possibles solucions**, **seleccionar les que compleixen unes determinades restriccions**, **triar la millor**.

La resolució de problemes per cerca exhaustiva sol comportar l'exploració d'espais molt grans de solucions, per la qual cosa resulta pràctica només per a instàncies petites del problema.

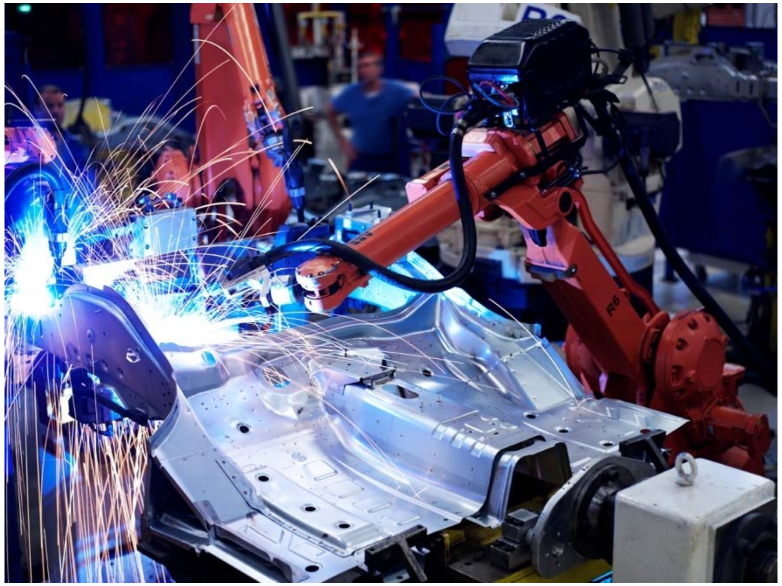
Cerca exhaustiva: TSP o el problema del viatjant de comerç.

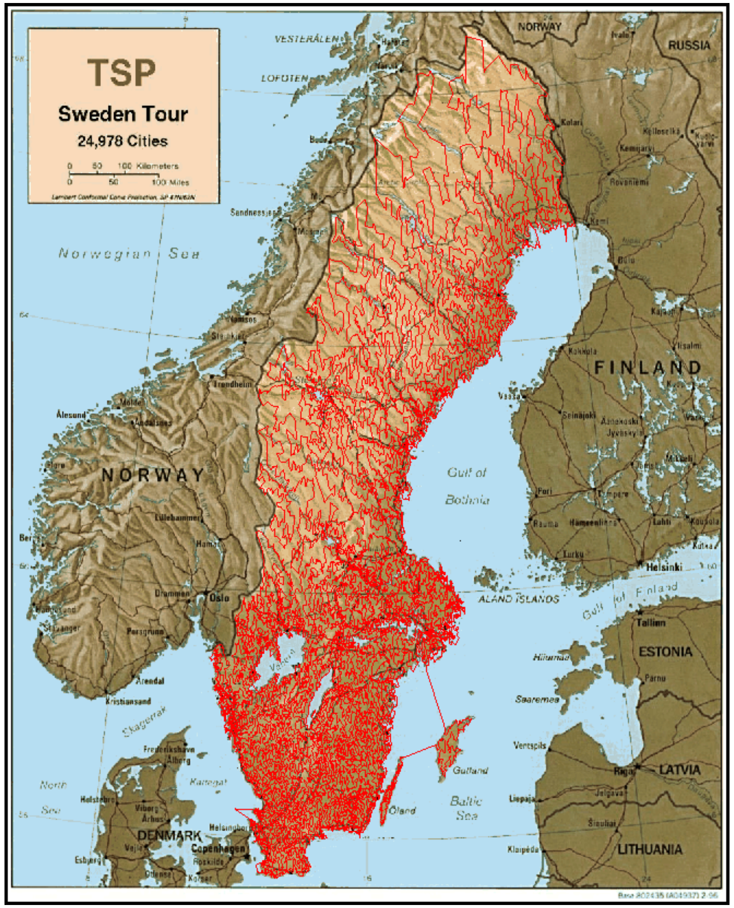
Donat un conjunt de **llocs** o ciutats, es tracta de trobar l'ordre a seguir per tal de tal que el **camí** fet pel viatjant de comerç passant per tots els llocs, des del punt de partida fins al punt d'arribada, sigui el més **curt** possible.

El problema del viatjant de comerç es presenta en moltes aplicacions pràctiques, per exemple en la **planificació d'un viatge, en logística o en el disseny del microxips**.

Encara apareix més freqüentment com a subproblema, per exemple en el problema de la distribució de mercaderies, en el problema de la planificació de la ruta per donar servei als clients o en la seqüenciació del genoma.

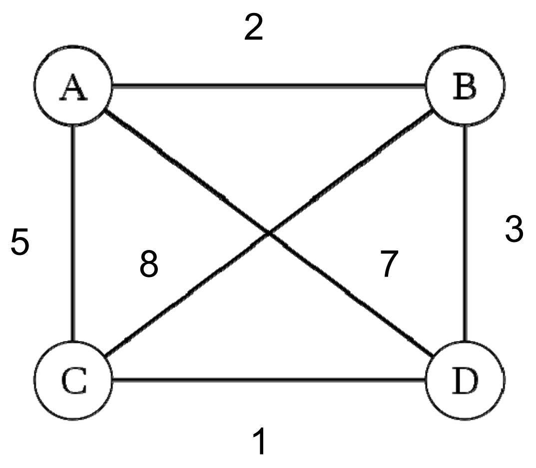
Cerca exhaustiva: TSP o el problema del viatjant de comerç.





Cerca exhaustiva:TSP o el problema del viatjant de comerç.

El problema del viatjant de comerç es pot modelitzar amb l'ajuda d'un graf utilitzant els **vèrtex i les arestes.**



Les ciutats estan representades pels vèrtexs $v_1,...,v_n$ i les carreteres entre les ciutats per les arestes a_{ij} entre dos vèrtexs v_i i v_j.

Cada aresta a_{ij} té una determinada longitud que, depenent del context, signifiquen la longitud geogràfica d'una connexió, el temps emprat en el recorregut o les despeses de viatge.

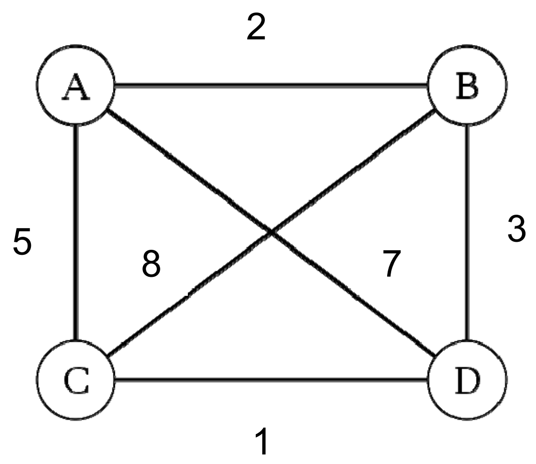
	A	B	C	D
A	-	2	5	7
B		-	8	3
C			-	1
D				-

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

Una ***ruta*** (també conegut com **circuit hamiltonià**) és un circuit que passa per tots els vèrtexs i en el que cada vèrtex surt exactament una vegada. (= una seqüència de n vèrtexs diferents = una seqüència de n+1 vèrtexs que comencen i acaben al mateix vèrtex).

Suposem que el viatjant va en avió I pot anar de qualsevol ciutat a qualsevol ciutat

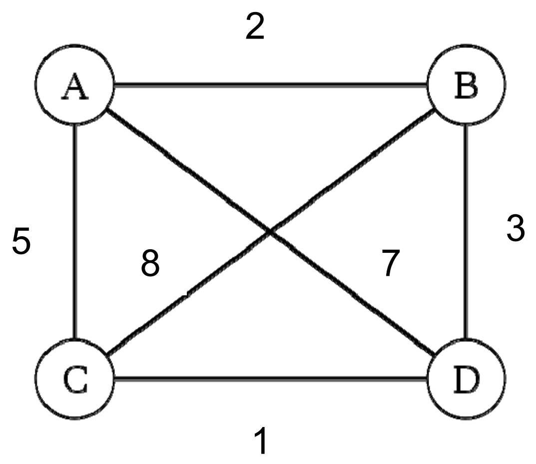
L'objectiu és trobar la ruta més curta possible.



Ruta possible: A → D → B → C

A → D → B → C → A

Cerca exhaustiva:TSP o el problema del viatjant de comerç.



Generar totes les **possibles rutes** és el mateix que generar totes les possibles **permutacions** dels vèrtexs del mig.

Rutes possibles					Cost
A	B	C	D	A	2+8+1+7=18
A	B	D	C	A	**2+3+1+5=11**
A	C	B	D	A	5+8+3+7=23
A	C	D	B	A	**5+1+3+2=11**
A	D	B	C	A	7+3+8+5=23
A	D	C	B	A	7+1+8+2=18

Òptima

Òptima

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

De fet, podem obviar la meitat de les rutes: B-C-D = D-C-B!

Per tant, podem triar dues ciutats del mig (per exemple B i C) i tenir en compte només les permutacions on B precedeix C (= aquest petit truc defineix la direcció de la ruta!).

Tot i això, el nombre de rutes és (n-1)!/2....

Com generem les possibles **permutacions**?

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

Algorisme de **Johnson-Trotter** per generar permutacions:

Sense perdre generalitat, associa cada símbol a un enter.

Aquest algorisme assigna una direcció a cada element d'una permutació:

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$$

L'element *k* es diu *mòbil* si l'element contigu en la direcció que assenyala és menor que ell (a l'exemple, 3 i 4 són *mòbils*).

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

Algorisme de **Johnson-Trotter** per generar permutacions:

Entrada: una llista d'enters.
Sortida: una llista amb totes les permutacions.

- Inicialitza la primera permutació amb tots els elements 1,2,...,*n* mòbils: tots ⟵
- Mentre hi hagi un element mòbil:
  1. Troba l'enter mòbil *k* més gran
  2. Intercanvia *k* i l'element adjacent al qual senyala
  3. Inverteix la direcció de tots els elements que són més grans que *k*
  4. Afegeix la permutació a la llista.

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

Exemple:

<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

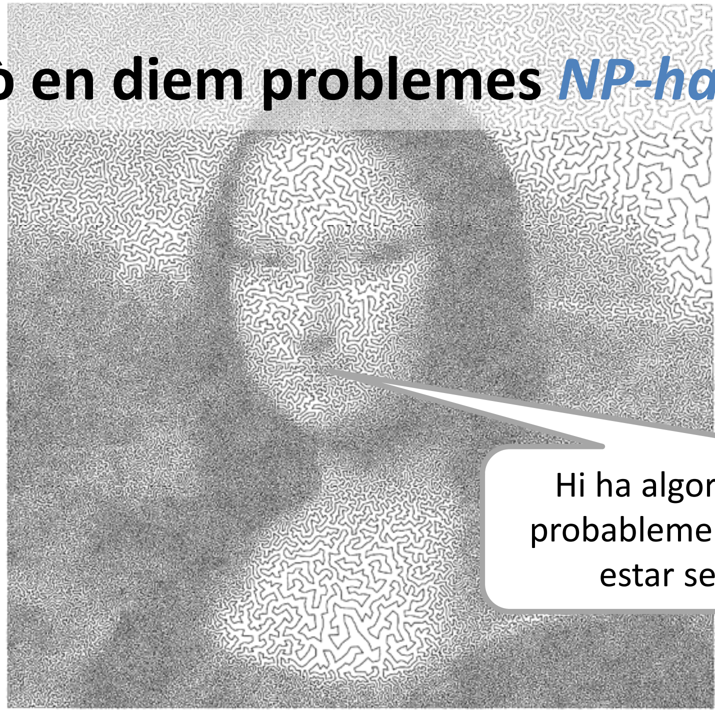
4>	<2	<1	3>
<2	4>	<1	3>
<2	<1	4>	3>
<2	<1	3>	4>

- Evidentment, té una complexitat $O(n!)$
- En el nostre cas, 24 solucions.

Cerca exhaustiva:TSP o el problema del viatjant de comerç.

El problema del viatjant de comerç no té una **solució exacta** més eficient que la cerca exhaustiva: no es coneix cap algorisme exacte en temps polinòmic.

D'això en diem problemes *NP-hard*.



Hi ha algorismes que troben solucions probablement bones, tot i que no podem estar segurs que siguin òptimes.

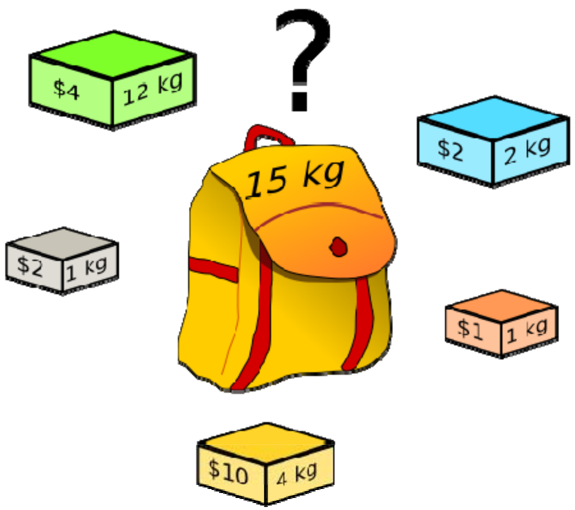
We can distribute cities with a density that locally approximates the darkness of a source image, and pass the cities to a program that finds a TSP tour.

100.000 punts.

Cerca exhaustiva: El problema de la motxilla.

El **problema de la motxilla**, altrament dit **KP** (en anglès, *Knapsack Problem*) és un problema d'optimització combinatòria.

Modelitza una situació anàloga al fet d'omplir una motxilla, en la que no es pot posar més d'un cert pes, amb tot o una part d'un conjunt d'objectes. Aquests objectes tenen un pes i un valor determinat.
Els objectes que es posen dins la motxilla han de maximitzar el valor total sense sobrepassar el pes màxim.



Cerca exhaustiva: El problema de la motxilla.

Com es generen les possibles solucions?

Generar les possibles solucions d'aquest problema és el mateix que generar tots els **possibles subconjunts d'un conjunt** ($O(2^n)$).

Després podríem seleccionar les que "caben" a la motxilla, i per últim, entre les que hi caben, quina és la més valuosa.

Cerca exhaustiva: El problema de la motxilla.

És útil per modelar:

• sistemes de suport a la gestió del portafolis : per equilibrar la selecció i la diversificació amb l'objectiu de trobar el millor equilibri entre el rendiment i el risc d'un capital col·locat en diferents actius financers (accions...);
• en la càrrega d'un vaixell o d'un avió: tot l'equipatge que es pot portar sense sobrepès;
• en el tall dels materials: per minimitzar les pèrdues degudes als talls (de diferents mides) realitzats en barres de ferro;
• en el problema de *bin packing*.

Cerca exhaustiva: El problema de la motxilla.

Com generem els subconjunts d'un conjunt?

Ens basarem en fer una correspondència entre els 2^n subconjunts d'un conjunt $A=\{a_1,...,a_n\}$ de n elements i els 2^n *strings* de bits de longitud n, $b_1,...,b_n$

Hi ha 2^n possibles subconjunts: un element hi pertany o no...

Per exemple, si n=3, el *string* 000 representa el conjunt buit, el 111 correspon al conjunt sencer, i 101 és el subconjunt format pel primer i el tercer element.

Cerca exhaustiva: El problema de la <u>motxilla</u>.

Com generem els subconjunts d’un conjunt?

Feta aquesta associació, podem generar tots els subconjunts d’un conjunt de n elements generant de forma successiva els nombres binaris des de 0 fins a 2^n-1, posant els 0’s que siguin necessaris al davant:

000 001 010 011 100 101 110 111

Cerca exhaustiva: El problema de la motxilla.

El problema de la motxilla no té una solució exacta més eficient que la cerca exhaustiva: **no es coneix cap algorisme exacte en temps polinòmic.**

D’això en diem problemes *NP-hard*.