

Tema 5: Algorismes i força bruta

Resum del tema 5

- Ordenar
 - Ordenació per selecció
- Algorismes intractables: cas fort (NP-hard) i cas dèbil
 - Cerca exhaustiva
 - TSP
 - Knapsack

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, sense cap mena d'optimització. Exemples:

- Calcular $a^n \bmod m$, per ($a > 0, n \geq 0$), calculant primer a^n i després passant aquest resultat a $\bmod n$. Ja hem vist que hi havia millors maneres de fer-ho!
- Calcular $n!$ com $n * (n-1) * \dots * 1$. En aquest cas no sabem fer-ho millor.
- Multiplicar dues matrius X i Y aplicant la definició de multiplicació de matrius:

```
for i in range(len(X)):
    for j in range(len(Y[0])):
        for k in range(len(Y)):
            result[i][j] += X[i][k] * Y[k][j]
```

El problema de l'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!

Hi ha molts algorismes d'ordenació. Anem a veure'n un basat en la força bruta.

Algorismes d'ordenació d'una llista

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

Ordenació per selecció

L'algorisme d'ordenació per selecció segueix l'analogia d'ordenació ingènua d'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 $A[i+1..n-1]$ i el canviem per $A[i]$.

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

Ordenació per selecció

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

```
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]
```

L'operació més important és una comparació:

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

I el nombre de vegades que s'executa és:

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

Evidentment l'algorisme és **quadràtic**, $O(n^2)$, tot i que **només fem $O(n)$ intercanvis** a la llista.

Avís Important

L'ordenació per selecció no és un bon mètode d'ordenació perquè hi ha altres algorismes de complexitat $O(n \log n)$, tal com hem vista a la taula anterior.

No useu mai un algorisme $O(n^2)$ per ordenar (bubble sort, insertion sort, ...). Aquí només els veiem com a exemple algorísmic, no perquè siguin útils.

Conceptes Algorísmics

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 [indecidibilitat de l'aturada d'un programa](#)).
- Cas dèbil: **No es coneix cap algorisme eficient** per resoldre el problema (p.e. la factorització de nombres enters grans) i per tant no són capaços de trobar-ne la solució. Normalment s'enten que un algorisme *eficient* ha de ser com a màxim de tipus polinòmic.

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

Una forma d'afrontar la intractabilitat són els **algorismes aproximats**: algorismes que poden trobar amb alta probabilitat una *bona* solució del problema però que no ens poden mai assegurar que sigui la millor.

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:

- generar totes les possibles solucions,
- seleccionar les que compleixen unes determinades restriccions (si és necessari),
- 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.

TSP o el problema del viatjant de comerç.

El problema TSP es pot resoldre per cerca exhaustiva si tenim un conjunt reduït de ciutats.

Definició: 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.

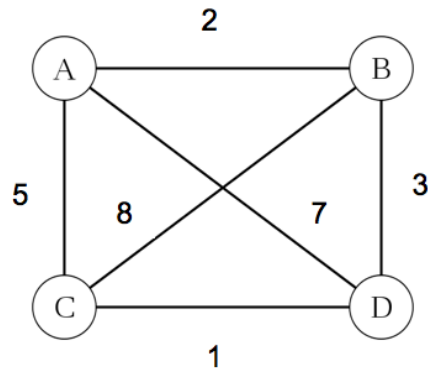
Encara apareix més freqüentment com a subproblema, per exemple en el problema de la distribució de mercaderies, en el problema de la planificació de la ruta per donar servei als clients o en la seqüenciació del genoma.

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, \dots, 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, significa 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				-

TSP o el problema del viatjant de comerç.

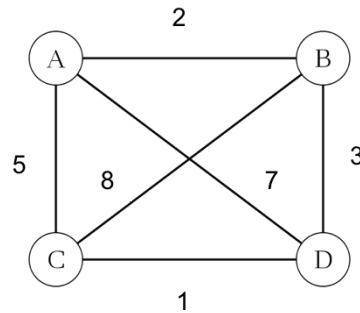
Una ruta (també coneguda com **circuit hamiltonià**) és un circuit que passa per tots els vèrtexs i en el que cada vèrtex surt exactament una vegada.

Per definir l'algorisme farem servir aquesta observació: una seqüència de n vèrtexs diferents = una seqüència de $n+1$ vèrtexs que comencen i acaben al mateix vèrtex.

L'objectiu és trobar la ruta més curta possible.

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. Recordeu que el nombre de permutacions de n elements és $n!$.



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

TSP o el problema del viatjant de comerç.

De fet, podem fer-ho una mica millor si ens adonem que podem obviar la meitat de les rutes: la ruta B-C-D-B té la mateixa longitud que la ruta B-D-C-B

Per tant, podem triar dues ciutats del mig (per exemple D i C) i tenir en compte només les permutacions on D 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?

Algorisme de Johnson-Trotter

Algorisme de Johnson-Trotter per generar permutacions:

1. Primer associa un símbol a cada enter.
2. Després assigna una direcció a cada símbol:

→ ← → ←
3 2 4 1

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

Algorisme de Johnson-Trotter

Entrada: una llista d'enters.

Sortida: una llista amb totes les permutacions.

1. Inicialitza la primera permutació amb tots els elements $1, 2, \dots, n$ mòbils: tots amb una fletxa mirant a l'esquerra.
2. Mentre hi hagi un element mòbil:
 - 2.1 Troba l'enter mòbil k més gran
 - 2.2 Intercanvia k i l'element adjacent al qual assenyalava
 - 2.3 Inverteix la direcció de tots els elements que són més grans que k
 - 2.4 Afegeix la permutació a la llista.

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

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.

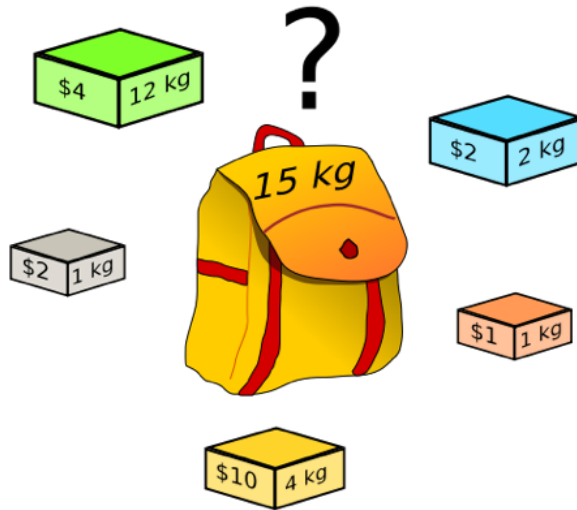
D'això en diem problemes **NP-hard**.

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

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** dels objectes transportats sense sobrepassar el pes màxim.



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**. El nombre de subconjunts diferents d'un conjunt de n elements és $O(2^n)$.

Un cop generats els subconjunts podríem seleccionar les que “cabem” a la motxilla, i per últim, entre les que hi cabem, quina és la més valuosa.

El problema de la motxilla.

És útil per modelar:

- sistemes de suport a la gestió del portafolis financers: 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 emplenat de contenidors o *bin packing*.

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 $A = \{a_1, \dots, a_n\}$ de n elements i els 2^n strings de bits de longitud n , b_1, \dots, b_n .

El nombre de subconjunts d'un conjunt és 2^n : cada un dels subconjunts es defineix pel fet que un determinat elements hi pertany o no, i per tant es pot representar com un nombre binari de n elements.

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.

El problema de la motxilla.

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

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.

Possibles preguntes d'exàmen relacionades amb el tema 5

1. Quin cost té un *bon* algorisme d'ordenació?
2. Quin cost té l'ordenació per selecció?
3. Perquè diem que el problema del viatjant de comerç és NP-hard?
4. Es pot trobar una solució exacta millor al problema de la motxilla?