Tema 6. Cerca exhaustiva

Estructures de Dades i Algorismes

FIB

Transparències d' Antoni Lozano
(amb edicions menors d'altres professors)

Q1 2020 – 21

1 Algorismes de força bruta

2 Backtracking
- Cadenes amb uns
- Permutacions
- Algorisme genèric
- Les *n* reines
- Quadrats llatins
- Els salts de cavall
- La motxilla
- El viatjant de comerç
- Graf Hamiltonià

1 Algorismes de força bruta

2 Backtracking
- Cadenes amb uns
- Permutacions
- Algorisme genèric
- Les *n* reines
- Quadrats llatins
- Els salts de cavall
- La motxilla
- El viatjant de comerç
- Graf Hamiltonià

- Molts problemes consisteixen en, donat un conjunt de restriccions, trobar un objecte que les satisfà (una solució)
- Per exemple, resoldre un sudoku.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- Hi ha variacions:
  - trobar/comptar totes les solucions
  - trobar la millor de totes les solucions (solució òptima)
  - etc.

Algorismes de força bruta

Sovint l'única forma de resoldre aquests problemes és provar totes les possibilitats. D'això en diem força bruta o cerca exhaustiva:

- Acostuma a ser exponencial.
- Pot ser lenta, però millor que res...
- Pot arribar a ser pràctica amb entrades petites.
- Es pot ajudar d'altres tècniques
  (com dividir i vèncer, algorismes voraços, etc.).

Algorismes de força bruta

Suposem que volem escriure totes les cadenes de zeros i uns de mida n.

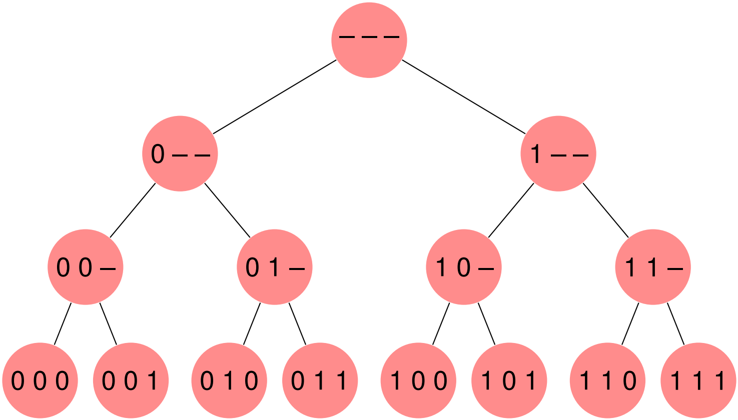
Tenim un procediment `escriu(vector<int>& A)` que escriu el vector `A`. Llavors es crida `binari(0, A)`, on $n =$ `A.size()` i `binari` es defineix així:

```
// i es la seguent posicio del vector A que assignarem
void binari(vector<int>& A, int i) {
  if (i == A.size()) escriu(A);     // cas base
  else {                            // cas inductiu
    A[i] = 0; binari(A,i+1);
    A[i] = 1; binari(A,i+1);
} }

void binari(int n) {
  vector<int> A(n);
  binari(A,0);
}
```

Algorismes de força bruta

Per a $n = 3$, s'obté el següent arbre de recursió:



Les fulles són solucions.

Les arestes indiquen com estenem cada solució parcial.

Els nodes interns són solucions parcials.

Quin cost té la cerca exhaustiva?

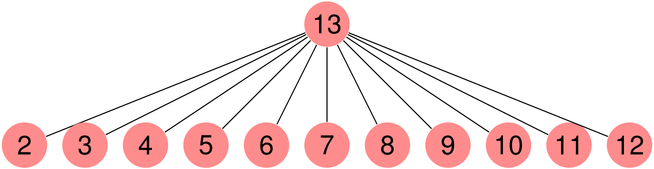
- si hi ha un arbre o graf implícit, normalment són exponencials
- si el graf ve donat a l'entrada, són cerques polinòmiques

  Per exemple, les cerques en profunditat i amplada en grafs també són cerques exhaustives

Exemple: primers

```
bool es_primer (Integer x) {
    if (x <= 1) return false ;
    for (int i = 2; i < x; ++i)
        if (x % i == 0) return false ;
    return true; }
```

Nombre màxim d'iteracions: $(x - 1) - 2 + 1 = x - 2$.
Cost en funció de x: $\Theta(x)$
Cost en funció de $n = |x|$: $\Theta(2^n)$.



Arbre implícit per a $x = 13$

1 Algorismes de força bruta

2 Backtracking
- Cadenes amb uns
- Permutacions
- Algorisme genèric
- Les *n* reines
- Quadrats llatins
- Els salts de cavall
- La motxilla
- El viatjant de comerç
- Graf Hamiltonià

Backtracking

Un algorisme de backtracking funciona com una cerca exhaustiva, però no continua quan veu que una solució parcial no es pot estendre a una solució

Els algorismes de backtracking són més eficients que una simple cerca exhaustiva, però el cost és sovint encara exponencial.

En català, *backtracking* es tradueix per:

- tornada enrere
- cerca amb retrocés

Backtracking

Exemple: moblar un pis

- Estratègia de força bruta: provar totes les configuracions dels mobles en tots els espais.
- L'estratègia de backtracking usa que:
  - cada moble acostuma a anar a un espai concret
    (no posarem el sofà a la cuina)
  - hi ha mobles que van junts
    (cadires i taula, llit i tauletes)
  - si una subdistribució no és satisfactòria,
    no considerarem la distribució que la conté
    (si no ens agrada posar un moble davant d'una finestra,
    ja no explorarem a partir d'aquí)

Backtracking – Cadenes amb uns

Problema

Volem escriure totes les cadenes de zeros i uns de mida n que contenen exactament k uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?

Backtracking – Cadenes amb uns

Problema

Volem escriure totes les cadenes de zeros i uns de mida n que contenen exactament k uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?
  - D'esquerra a dreta.

Backtracking – Cadenes amb uns

Problema

Volem escriure totes les cadenes de zeros i uns de mida n que contenen exactament k uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?
  - D'esquerra a dreta.
- **Segona pregunta:** com representarem una cadena parcial en C++?

Backtracking – Cadenes amb uns

Problema

Volem escriure totes les cadenes de zeros i uns de mida n que contenen exactament k uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?
  - D'esquerra a dreta.
- **Segona pregunta:** com representarem una cadena parcial en C++?
  - Tindrem un vector A de mida n i un enter idx que indicarà quina és la primera posició no omplerta.

A	0	1	1	0	?	?	?
					$\uparrow$		
					idx		

Backtracking – Cadenes amb uns

Problema

Volem escriure totes les cadenes de zeros i uns de mida n que contenen exactament k uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?
  - D'esquerra a dreta.
- **Segona pregunta:** com representarem una cadena parcial en C++?
  - Tindrem un vector A de mida n i un enter idx que indicarà quina és la primera posició no omplerta.

A	0	1	1	0	?	?	?
					$\uparrow$		
					idx		

- **Tercera pregunta:** donada una cadena parcial, quins candidats tinc per omplir la posició idx?

Problema

Volem escriure totes les cadenes de zeros i uns de mida *n* que contenen exactament *k* uns.

- En un moment donat de l'algorisme, tindrem una cadena parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la cadena?
  - D'esquerra a dreta.
- **Segona pregunta:** com representarem una cadena parcial en C++?
  - Tindrem un vector *A* de mida *n* i un enter *idx* que indicarà quina és la primera posició no omplerta.

A	0	1	1	0	?	?	?
					↑ *idx*		

- **Tercera pregunta:** donada una cadena parcial, quins candidats tinc per omplir la posició *idx*? El 0 i l'1.

Backtracking – Cadenes amb uns

```
// A: cadena parcial (mida n)
// idx: primera casella no omplerta de A
// k: nombre total d'1s que volem
void cadenes(vector<int>& A, int idx, int k) {
  if (idx == A.size()) {
    int c = 0;
    for (int x : A) c += x; // Compto els 1s
    if (c == k) escriu(A);
  }
  else {
    A[idx] = 0;  cadenes(A,idx+1,k);
    A[idx] = 1;  cadenes(A,idx+1,k);
} }

int main(){
  int n, k;      cin >> n >> k;
  vector<int> A(n);
  cadenes(A,0,k);
}
```

Backtracking – Cadenes amb uns

- Cal comptar cada vegada el nombre d'1s de la cadena *A*?
  - Podem mantenir, en cada moment, el nombre d'1s de la cadena parcial
  - Això implica passa un paràmetre més al procediment

Backtracking – Cadenes amb uns

```
// A: cadena parcial (mida n)
// idx: primera casella no omplerta de A
// u: nombre d'1s que hi ha en A[0...idx-1] (ja portem posats)
// k: nombre total d'1s que volem
void cadenes2(vector<int>& A, int idx, int u, int k) {
  if (idx == A.size()) {
    if (u == k) escriu(A);
  }
  else {
    A[idx] = 0;     cadenes2(A,idx+1,u,k);
    A[idx] = 1;     cadenes2(A,idx+1,u+1,k);
  }
}

int main(){
  int n, k;   cin >> n >> k;
  vector<int> A(n);
  cadenes2(A,0,0,k);    }
```

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 3$, què passa en la situació següent?

0	1	1	1	?	?	?

$$\begin{array}{ccccccc} & & & & \uparrow & & \\ & & & & idx & & \end{array}$$

Backtracking – Cadenes amb uns

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 3$, què passa en la situació següent?

0	1	1	1	?	?	?

$$\begin{array}{ccccccc} & & & & \uparrow & & \\ & & & & idx & & \end{array}$$

En la posició *idx* no hi podem posar un 1

Backtracking – Cadenes amb uns

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament *k* 1s?
- Si $k = 3$, què passa en la situació següent?

0	1	1	1	?	?	?
				$\uparrow$		
				idx		

En la posició *idx* no hi podem posar un 1

- Generalitzant, si portem *u* 1s posats i en total en volem *k*, només podem posar 1 a la posició *idx* si $u < k$
- Acabem de dissenyar una poda.

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 5$, què passa en la situació següent?

0	1	0	1	?	?	?

$$\uparrow$$
$$idx$$

Backtracking – Cadenes amb uns

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 5$, què passa en la situació següent?

0	1	0	1	?	?	?
				$\uparrow$ *idx*		

En la posició *idx* no hi podem posar un 0

Backtracking – Cadenes amb uns

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 5$, què passa en la situació següent?

0	1	0	1	?	?	?
				$\uparrow$		
				idx		

En la posició *idx* no hi podem posar un 0

- Generalitzant, si volem k uns, necessitarem $n - k$ zeros. Si portem z zeros, només podem posar un 0 a la posició *idx* si $z < n - k$.
- Acabem de dissenyar una altra poda.

Backtracking – Cadenes amb uns

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 5$, què passa en la situació següent?

0	1	0	1	?	?	?

$$\uparrow$$
$$idx$$

En la posició *idx* no hi podem posar un 0

- Generalitzant, si volem k uns, necessitarem $n - k$ zeros. Si portem z zeros, només podem posar un 0 a la posició *idx* si $z < n - k$.
- Acabem de dissenyar una altra poda.
- **Cinquena pregunta:** com podem implementar aquestes podes de manera eficient?

- **Quarta pregunta:** podem detectar situacions en les que una cadena parcial no pot extendre's a una cadena total amb exactament k 1s?
- Si $k = 5$, què passa en la situació següent?

0	1	0	1	?	?	?
				$\uparrow$ *idx*		

En la posició *idx* no hi podem posar un 0

- Generalitzant, si volem k uns, necessitarem $n - k$ zeros. Si portem z zeros, només podem posar un 0 a la posició *idx* si $z < n - k$.
- Acabem de dissenyar una altra poda.
- **Cinquena pregunta:** com podem implementar aquestes podes de manera eficient?
- Una manera senzila és mantenir també el nombre de zeros que ja hem posat.

Backtracking – Cadenes amb uns

```
// A: cadena parcial (mida n)
// idx: primera casella no omplerta de A
// u: nombre d'1s que hi ha en A[0...idx-1]  (ja portem posats)
// z: nombre de 0s que hi ha en A[0...idx-1] (ja portem posats)
// k: nombre total d'1s que volem
void cadenes3(vector<int>& A, int idx, int z, int u, int k) {
  if (idx == A.size()) escriu(A);
  else {
    if (z < A.size() - k) { // no tots els 0s posats
      A[idx] = 0;  cadenes3(A,idx+1,z+1,u,k); }

    if (u < k) { // no tots els 1s posats
      A[idx] = 1;  cadenes3(A,idx+1,z,u+1,k); }
  }
}

int main(){
  int n, k;  cin >> n >> k;
  vector<int> A(n);
  cadenes3(A,0,0,0,k); }
```

Backtracking – Cadenes amb uns

Com es comparen les tres solucions? (només comptant solucions, sense escriure)

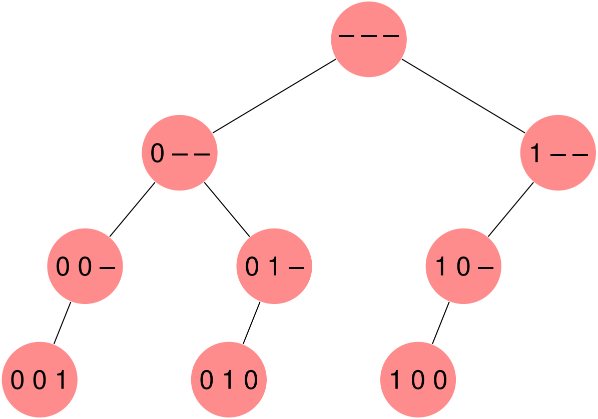
Per exemple, si $n = 30$

Algorisme	Segons (k=2)	Segons (k=8)	Segons (k=15)
cadenes1	13.5	13.5	13.5
cadenes2	4	4	4
strings3	0.007	0.08	1.5

No obstant, si prenem $k = n/2$ veurem que la tercera solució també és exponencial en n, ja que ha de comptar $\binom{n}{n/2}$ cadenes.

n	10	16	22	28	34
$\binom{n}{n/2}$	252	12,870	705,432	40,116,600	2,333,606,220

Per a $n = 3$ i $k = 1$, s'obté l'arbre de recursió:



És millor que generar totes les possibilitats i després comprovar els uns.

Però encara hi ha un nombre exponencial de nodes.

Exemple: permutacions de *n* elements

Quines són les permutacions dels naturals $\{1, \ldots, n\}$?

- Hi ha n possibilitats per al primer.
- Fixat el primer, hi ha $n - 1$ possibilitats per al segon.
- Repetint el raonament, obtenim

$$\prod_{k=1}^{n} k = n!.$$

Per $n = 4$, tenim les permutacions:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Backtracking – Permutacions

- En un moment donat de l'algorisme, tindrem una permutació parcial i caldrà extendre-la de totes les maneres possibles.
- **Primera pregunta:** com omplirem la permutació?
  - D'esquerra a dreta.
- **Segona pregunta:** com representarem una permutació parcial en C++?
  - Tindrem un vector *A* de mida *n* i un enter *idx* que indicarà quina és la primera posició no omplerta.

A	3	4	6	1	?	?	?
					$\uparrow$		
					idx		

- **Tercera pregunta:** donada una permutació parcial *A*, quins candidats tinc per omplir la posició *idx*? Els elements de $\{1, 2, \ldots, n\}$ no presents a *A*.

Backtracking – Permutacions

```
// n: volem permutacions del nombres {1,2,...,n}
// A: permutacio parcial (mida n)
// idx: primera casella no omplerta de A
void escriu_permutacions1(int n, vector<int>& A, int idx) {
  if (idx == A.size()) escriu(A);
  else {
    for (int k = 1; k <= n; ++k) {
      bool usat = false; // Determinem si k ja ha estat usat
      for (int i = 0; i < idx and not usat; ++i)
        usat = (A[i] == k);
      if (not usat) {
        A[idx] = k;
        escriu_permutacions1(n,A,idx+1);
      }  }  }  }

int main() {
  int n; cin >> n;
  vector<int> A(n);
  escriu_permutacions1(n,A,0);
}
```

Backtracking – Permutacions

- Podem evitar el càlcul de *usat* cada vegada?
- Fàcilment podem mantenir aquesta informació amb un vector *usat* tal que $usat[k]$ és cert sii el nombre k ja apareix a la permutació parcial.
- **Compte:** aquesta informació cal mantenir-la també sota backtrack.
- Si només comptem permutacions de 12 elements (no escrivim), aquesta millora permet passar de 112 segons a 30 segons.

```
void escriu_permutacions2(int n, vector<int>& A, int idx,
    vector<bool>& usat) {
  if (idx == A.size()) escriu(A);
  else {
    for (int k = 1; k <= n; ++k) {
      if (not usat[k]) {
        A[idx] = k;
        usat[k] = true;
        escriu_permutacions2(n,A,idx+1,usat);
        usat[k] = false; // restaurem sota backtrack
      }
    }
  }
}

int main() {
  int n; cin >> n;
  vector<int> A(n);
  vector<bool> usat(n+1,false);
  escriu_permutacions2(n,A,0,usat); }
```

- **Quarta pregunta:** podem detectar situacions en les que una permutació parcial no pugui extendre's a una permutació total? No
- Per tant, no podem fer cap poda. De fet, ja l'hem fet quan hem seleccionat els candidats per la posició *idx*.

Algorisme genèric

Es pot definir un algorisme genèric de tornada enrere:

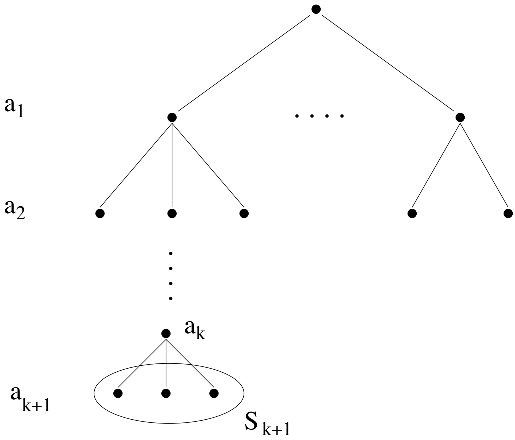
- L'espai de solucions (parcials) d'un problema s'acostuma a organitzar en forma d'arbre de configuracions.
- Cada node o configuració de l'arbre es representa amb un vector

  $$A = (a_1, a_2, \ldots, a_k)$$

  que conté les tries ja fetes.
- El vector A s'amplia en la fase *avançar* triant un a_{k+1} d'un conjunt de candidats S_{k+1} (explorar en profunditat).
- A es redueix en la fase *retrocedir* (backtrack).

Algorisme genèric

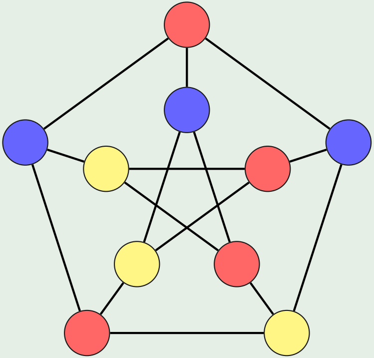
Un algorisme de tornada enrere és sovint una cerca en profunditat en un arbre de configuracions:



Algorisme genèric

Exemple: 3-Colorabilitat

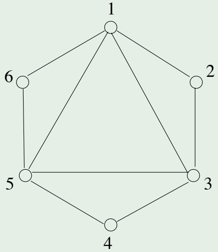
El problema de la 3-colorabilitat consisteix en
decidir si es pot assignar un color a cada vèrtex (d'un total de 3)
de manera que els adjacents tinguin colors diferents.



Una 3-coloració del graf de Petersen

3-colorabilitat

Donat un graf, per exemple



- Les configuracions seran assignacions parcials de colors, és a dir,

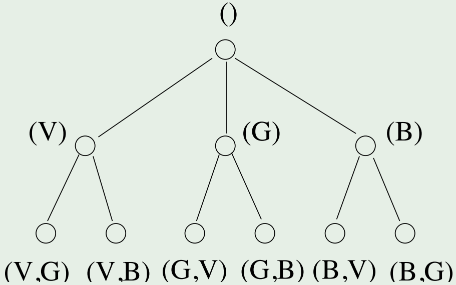
$$A = (a_1, a_2, \ldots, a_k)$$

representarà el fet que el vèrtex i s'acoloreix amb el color $a_i \in \{B, G, V\}$.

- El conjunt de candidats S_{k+1} per a a_{k+1} contindrà els colors compatibles amb els veïns que ja han estat acolorits.

Algorisme genèric

Els 3 primers nivells de l'arbre de configuracions serien:



Però si el que volem és trobar només una solució,

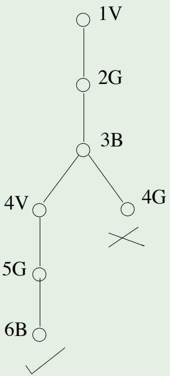
- es pot fixar un color per al vèrtex 1
- es pot fixar també un color per al vèrtex 2 sempre que sigui diferent
- qualsevol altra solució serà simètrica (ha d'assignar colors diferents)

Algorisme genèric

Fent la tria

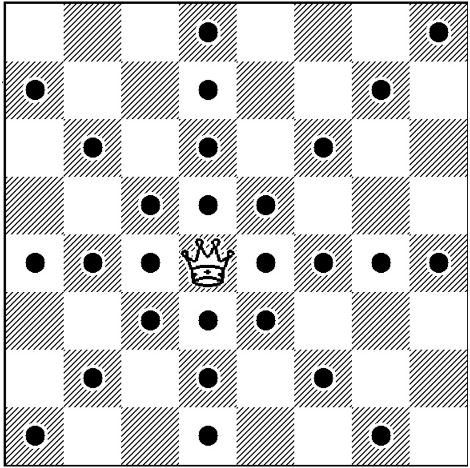
- $S_1 = \{V\}$
- $S_2 = \{G\}$

i definint $S_{k+1} = \{c \in \{V, G, B\} \mid \forall i \leq k \ \ (\{i, k+1\} \in E \Rightarrow c \neq a_i)\}$, s'obté l'arbre de configuracions

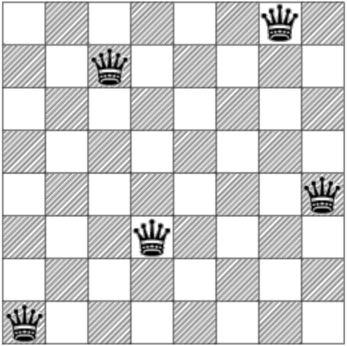


Les *n* reines

Moviments de la reina en el joc dels escacs:



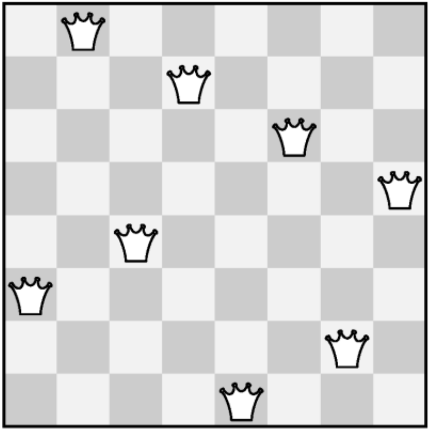
Quantes reines podem col·locar sobre un tauler sense que s'amenacin?
5? 6? 7? 8?



Les n reines

Problema de les 8 reines

Col·locar vuit reines en un tauler d'escacs sense que cap n'amenaci cap altra.



Les *n* reines

Estratègies de resolució per força bruta:

1. Triar 8 posicions diferents del tauler.

$$\binom{64}{8} = 4.426.165.368 \text{ configuracions}$$

2. Triar 8 posicions en files diferents.

$$8^8 = 16.777.216 \text{ configuracions}$$

3. Triar 8 posicions en files i columnes diferents.

$$8! = 40.320 \text{ configuracions}$$

Amb estratègies de backtracking més sofisticades encara es pot millorar més.

Les n reines

Considerarem el problema generalitzat de les n reines.

Problema de les n reines

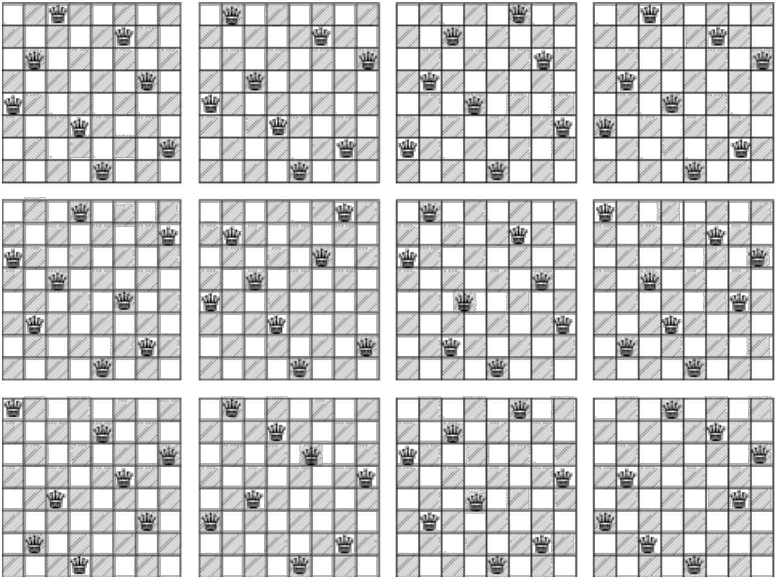
Col·locar n reines en un tauler $n \times n$ sense que cap n'amenaci cap altra.

Les n reines

Nombre de solucions no isomorfes (per rotació o reflexió) de les n reines per a $n \in \{1, \ldots, 10\}$:

n	solucions
1	1
2	0
3	0
4	1
5	2
6	1
7	6
8	12
9	46
10	92

Les 12 solucions no isomorfes per a $n = 8$



Primera implementació:

- troba totes les solucions
- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- cost en cas pitjor: $\Theta(n^n)$

Les n reines

Implementarem la posició de les reines amb un vector

```
vector<int> t;
```

que indicarà que la reina de la fila i és a la columna $t[i]$.

Les *n* reines

```
void escriu() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      cout << (t[i] == j ? "Q" : ".");
    cout << endl;
  }
  cout << endl;
}
```

Les n reines

Per saber si les reines de les files i i k ($i \neq j$) s'ataquen, recordem que les seves caselles són $(i, t[i])$ i $(k, t[k])$.

- columna, comprovem si $t[i] = t[k]$
- diagonal descendent ($\searrow$), comprovem si $t[i] - i = t[k] - k$
- diagonal ascendent ($\nearrow$), comprovem si $t[i] + i = t[k] + k$

	0	1	2	3	4
0	E	F	G	H	I
1	D	E	F	G	H
2	C	D	E	F	G
3	B	C	D	E	F
4	A	B	C	D	E

Diag A: $(4, 0)$
Diag B: $(3, 0), (4, 1)$
Diag C: $(2, 0), (3, 1), (4, 2)$
Diag D: $(1, 0), (2, 1), (3, 2), (4, 3)$
Diag E: $(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)$
Diag F: $(0, 1), (1, 2), (2, 3), (3, 4)$
Diag G: $(0, 2), (1, 3), (2, 4)$
Diag H: $(0, 3), (1, 4)$
Diag I: $(0, 4)$

Les n reines

Per saber si les reines de les files i i k ($i \neq j$) s'ataquen, recordem que les seves caselles són $(i, t[i])$ i $(k, t[k])$.

- columna, comprovem si $t[i] = t[k]$
- diagonal descendent ($\searrow$), comprovem si $t[i] - i = t[k] - k$
- diagonal ascendent ($\nearrow$), comprovem si $t[i] + i = t[k] + k$

	0	1	2	3	4
0	A	B	C	D	E
1	B	C	D	E	F
2	C	D	E	F	G
3	D	E	F	G	H
4	E	F	G	H	I

Diag A: $(0, 0)$
Diag B: $(1, 0), (1, 1)$
Diag C: $(2, 0), (1, 1), (0, 2)$
Diag D: $(3, 0), (2, 1), (1, 2), (0, 3)$
Diag E: $(4, 0), (3, 1), (2, 2), (1, 3), (0, 4)$
Diag F: $(4, 1), (3, 2), (2, 3), (1, 4)$
Diag G: $(4, 2), (3, 3), (2, 4)$
Diag H: $(4, 3), (3, 4)$
Diag I: $(4, 4)$

Les *n* reines

```
bool legal(int i) {
  for (int k = 0; k < i; ++k)
    if (t[k]     == t[i]     or
        t[k] - k == t[i] - i or
        t[k] + k == t[i] + i)
      return false;
  return true;
}

void reines(int i) {
  if (i == n) escriu();
  else
    for (int j = 0; j < n; ++j){ //j es col per reina de fila i
      t[i] = j;
      if (legal(i))
        reines(i+1);
    }
}
```

Les *n* reines

Segona implementació:

- troba totes les solucions
- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (es pugui estendre a una solució completa)
- amb marcatges
- cost en cas pitjor: $\Theta(n^n)$

Les n reines

ESTRATÈGIA DE MARCATGE:

- No cal marcar files usades (ona reina per fila per construcció)
- Fàcil marcar columnes usades (vector de Booleans de mida n)
- Diagonals?

	0	1	2	3	4
0	A	B	C	D	E
1	B	C	D	E	F
2	C	D	E	F	G
3	D	E	F	G	H
4	E	F	G	H	I

Diag A: $(0,0)$
Diag B: $(1,0),(1,1)$
Diag C: $(2,0),(1,1),(0,2)$
Diag D: $(3,0),(2,1),(1,2),(0,3)$
Diag E: $(4,0),(3,1),(2,2),(1,3),(0,4)$
Diag F: $(4,1),(3,2),(2,3),(1,4)$
Diag G: $(4,2),(3,3),(2,4)$
Diag H: $(4,3),(3,4)$
Diag I: $(4,4)$

Tenim 2n - 1 diagonals

Diagonal identificada amb $i+j$. Això dona nombres en $[0,2n-2]$.

ESTRATÈGIA DE MARCATGE:

- No cal marcar files usades (ona reina per fila per construcció)
- Fàcil marcar columnes usades (vector de Booleans de mida n)
- Diagonals?

	0	1	2	3	4
0	E	F	G	H	I
1	D	E	F	G	H
2	C	D	E	F	G
3	B	C	D	E	F
4	A	B	C	D	E

Diag A: $(4, 0)$
Diag B: $(3, 0), (4, 1)$
Diag C: $(2, 0), (3, 1), (4, 2)$
Diag D: $(1, 0), (2, 1), (3, 2), (4, 3)$
Diag E: $(0, 0), (1, 1), (2, 2), (3, 3)$
Diag F: $(0, 1), (1, 2), (2, 3), (3, 4)$
Diag G: $(0, 2), (1, 3), (2, 4)$
Diag H: $(0, 3), (1, 4)$.
Diag I: $(0, 4)$

Tenim 2n - 1 diagonals.

Diagonal identificada amb $i - j$. Això dona nombres en $[-(n-1), n-1]$.

Però usarem $i - j + (n - 1)$. Aixó dona nombres en $[0, 2n - 2]$.

Les *n* reines

```
#include <iostream>
#include <vector>

using namespace std;

int n;
vector<int> t;
// mc[j]  si ja hi ha reina a la columna j,
// md1[k] si ja hi ha reina a la diagonal i+j = k, etc.
vector<int> mc, md1, md2;

void reines(int i);

int main() {
  cin >> n;
  t = vector<int>(n);
  mc = vector<int>(n, false);
  md1 = md2 = vector<int>(2*n-1, false);
  reines(0);
}
```

Les *n* reines

```
int diag1(int i, int j) { return i+j; }
int diag2(int i, int j) { return i-j + n-1; }

void reines(int i) {
  if (i == n) escriu();
  else
    for (int j = 0; j < n; ++j)//j es col per reina de fila i
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        reines(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
}
```

Les *n* reines

Si només volem una solució, podem parar quan trobem la primera:

```
 // Diu si hi ha una solucio completant la solucio parcial
bool reines(int i) {
  if (i == n) {
    escriu();
    return true;
  }
  else {
    for (int j = 0; j < n; ++j)
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        if (reines(i+1)) return true;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return false;
  }
}
```

Les *n* reines

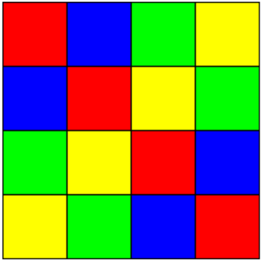
Si volem comptar solucions:

```
 // Diu quantes solucions hi ha completant la solucio parcial
int reines(int i) {
  if (i == n) {
    return 1;
  }
  else {
    int res = 0;
    for (int j = 0; j < n; ++j)
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        res += reines(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return res;
  }
}
```

Quadrats llatins

Un quadrat llatí és qualsevol quadrícula $n \times n$ omplerta amb n símbols diferents cadascun dels quals apareix un cop a cada fila i cada columna.

1	2	3
2	3	1
3	1	2



A	B
B	A

Quadrats llatins

Nombre de quadrats llatins $n \times n$ per a $n \in \{1, \ldots, 11\}$:

n	solucions
1	1
2	2
3	12
4	576
5	161280
6	812851200
7	61479419904000
8	108776032459082956800
9	5524751496156892842531225600
10	9982437658213039871725064756920320000
11	776966836171770144107444346734230682311065600000

Quadrats llatins

Problema dels quadrats llatins

Donat un n, trobar tots els quadrats llatins d'ordre n.

Quadrats llatins

Solució per tornada enrere amb marcatges.
Cost: $\mathcal{O}(n^{n^2})$.

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// q[i][j] == valor a la fila i, columna j
vector<vector<int>> q;

// f[i][v] si la fila i ja usa el valor v
vector<vector<bool>> f;

// c[j][v] si la columna j ja usa el valor v
vector<vector<bool>> c;
```

Quadrats llatins

```
void escriu() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      cout << q[i][j] << '\t';
    }
    cout << endl;
  }
  cout << endl;
}
```

Quadrats llatins

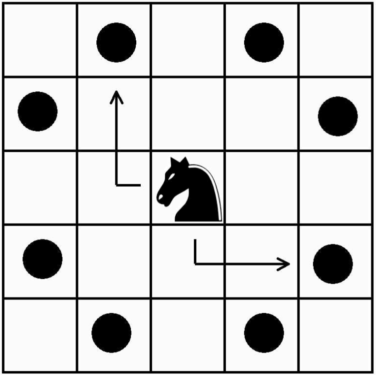
```
// Troba tots els quadrats llatins completant des de (i, j)
void quadrats_llatins(int i, int j) {
  if (i == n) return escriu();
  if (j == n) return quadrats_llatins(i+1, 0);
  for (int v = 0; v < n; ++v) {
    if (not f[i][v] and not c[j][v]) {
      f[i][v] = c[j][v] = true;
      q[i][j] = v;
      quadrats_llatins(i, j+1);
      f[i][v] = c[j][v] = false;
    }
  }
}

int main () {
  cin >> n;
  q = vector<vector<int>>(n, vector<int>(n));
  f = c = vector<vector<bool>>(n, vector<bool>(n, false));
  quadrats_llatins(0, 0);
}
```

Els salts de cavall

Salts de cavall (*knight's tour*)

Donat un tauler $n \times n$ i la posició d'una casella, trobar, si existeix, un recorregut del cavall d'escacs que visiti totes les caselles sense repeticions.



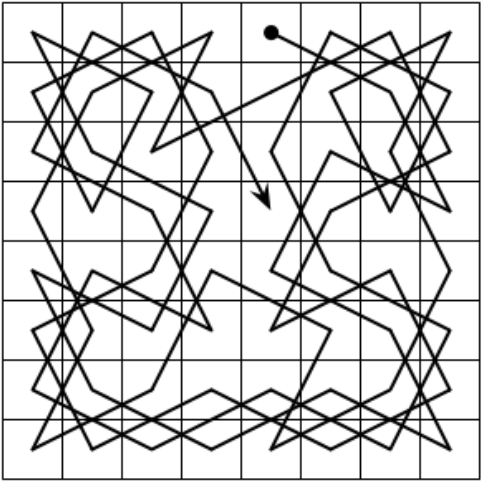
Els moviments del cavall

Els salts de cavall

El problema dels salts de cavall té una llarga tradició matemàtica. Prové de l'Índia del s. IX d.C. i el va treballar Euler al s. XVIII.

- Es considera en versió
  - tancada: l'inici i final estan a un salt de cavall
  - oberta: inici i final en posicions arbitràries
- Hi ha
  - 9.862 tours tancats no dirigits en un tauler 6×6
  - 13.267.364.410.532 tours tancats no dirigits en un tauler 8×8
- En alguns casos es poden trobar tours en temps polinòmic amb l'estratègia de dividir i vèncer.

Els salts de cavall



Una solució oberta per al tauler d'escacs

Els salts de cavall

Problema dels salts de cavall

Donat un tauler $n \times n$ i una casella (i, j),
volem trobar un tour obert que comenci en (i, j).

Solució per tornada enrere.

Els salts de cavall

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// t[i][j] == k quan en el salt k-esim arribem a (i,j)
// -1 si encara no hi hem arribat
vector<vector<int>> t;

// Si podem omplir el tauler des de (i, j) havent fet s salts
bool es_pot(int i, int j, int s);

int main() {
  int i, j;
  cin >> n >> i >> j;
  t = vector<vector<int>>(n, vector<int>(n, -1));
  cout << es_pot(i, j, 0) << endl;
}
```

Els salts de cavall

```
bool es_pot(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1 or
        es_pot(i+2, j-1, s+1) or es_pot(i+2, j+1, s+1) or
        es_pot(i+1, j+2, s+1) or es_pot(i-1, j+2, s+1) or
        es_pot(i-2, j+1, s+1) or es_pot(i-2, j-1, s+1) or
        es_pot(i-1, j-2, s+1) or es_pot(i+1, j-2, s+1))
      return true;
    t[i][j] = -1;
  }
  return false;
}
```

Els salts de cavall

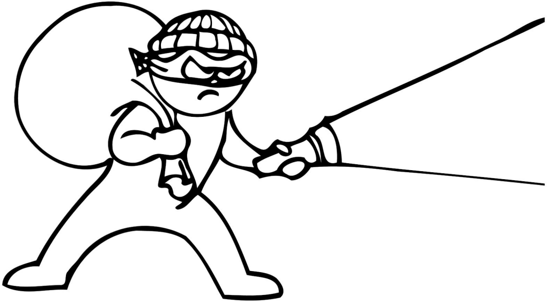
```
// Implementacio alternativa

vector<int> di = {1,  1, -1,  -1,  2,  2, -2,  -2};
vector<int> dj = {2, -2,  2,  -2,  1, -1,  1,  -1};

bool es_pot(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1) return true;
    for (int k = 0; k < 8; ++k)
      if (es_pot(i + di[k], j + dj[k], s+1))
        return true;
    t[i][j] = -1;
  }
  return false;
}
```

La motxilla

Suposem que un lladre vol entrar en una botiga i carregar al seu sac una combinació d'objectes amb el màxim valor total.



Com pot trobar la millor combinació fent ús de l'algorísmia?

La motxilla

En primer lloc cal:

- fer una llista amb els pesos i els valors dels objectes
- estimar fins a quin pes pot carregar en total com a màxim.

Ara només necessita un algorisme per actuar de pressa.

La motxilla

Problema de la motxilla

Donada una motxilla que pot carregar un pes C i n objectes amb

- pesos $p_1, p_2, \ldots, p_n$
- i valors $v_1, v_2, \ldots, v_n$

trobar una selecció $S \subseteq \{1, \ldots, n\}$ dels objectes

- amb valor $\sum_{i \in S} v_i$ màxim
- i que no superi la capacitat de la motxilla:

$$\sum_{i \in S} p_i \leq C.$$

La motxilla

Primera solució: podem quan superem la capacitat

```
#include <iostream>
#include <vector>

using namespace std;

int c;            // Capacitat
int n;            // Nombre d'objectes
vector<int> p;    // Pesos
vector<int> v;    // Valors
vector<int> s;    // Solucio

int bv = -1;      // Millor valor   fins ara
vector<int> bs; // Millor solucio fins ara
```

La motxilla

```
void opt(int k, int spp, int svp) { // spp: suma pesos parcial
  if (spp > c) return; // Capacitat excedida: no continuem
  if (k == n) {
    if (svp > bv) { // Millorem la solucio que teniem fins ara
      bs = s;
      bv = svp;
    }
    return;
  }
  s[k] = 0; opt(k+1, spp,        svp       ); // Deixem obj. k
  s[k] = 1; opt(k+1, spp + p[k], svp + v[k]); // Agafem obj. k
}

int main() {
  cin >> c >> n;
  p = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0);
  cout << bv << endl; }
```

La motxilla

Segona solució: podem quan superem la capacitat,
i quan ja no podem superar el millor cost trobat fins ara (branch & bound)

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int c;            // Capacitat
int n;            // Nombre d'objectes
vector<int> p;    // Pesos
vector<int> v;    // Valors
vector<int> s;    // Solucio

int bv = -1;      // Millor valor   fins ara
vector<int> bs;   // Millor solucio fins ara
```

La motxilla

```
//svr: suma valors restants (dels objectes per processar)
void opt(int k, int spp, int svp, int svr) {
  if (spp > c  or  svp + svr <= bv) return;
  if (k == n) {
    bs = s;
    bv = svp;
    return;
  }
  s[k] = 0; opt(k+1, spp,        svp,        svr - v[k]);
  s[k] = 1; opt(k+1, spp + p[k], svp + v[k], svr - v[k]);
}

int main() {
  cin >> c >> n;
  p = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0, accumulate(v.begin(), v.end(), 0));
  cout << bv << endl;
}
```

El viatjant de comerç

El problema del viatjant de comerç (travelling salesman) consisteix en, donada una xarxa de ciutats, trobar l'ordre en què visitar-les de forma que:

- es comença i s'acaba a la ciutat del viatjant
- es passa per la resta de ciutats exactament un cop, i
- la distància total recorreguda és la més curta possible

Ruta òptima d'un viatjant passant per les 15 ciutats més grans d'Alemanya.

Font: `upload.wikimedia.org/wikipedia/commons/c/c4/TSP_Deutschland_3.png`

El viatjant de comerç

- És un dels problemes d'optimització combinatòria més estudiats
- És important en informàtica teòrica
- Té aplicacions pràctiques en
  - planificació
  - logística
  - fabricació de microchips
  - seqüenciació d'ADN
  - astronomia
  - ...

El viatjant de comerç

```
class Viatjant {

  int n;                 // nombre de ciutats
  matriu<int> M;         // la matriu de distancies

  vector<int> s;         // seguent de cada vertex
                         // (−1 si encara no utilitzat)
                         // s es la solucio parcial que construim

  vector<int> sol;       // millor solucio trobada fins ara
  double millor_cost;    // cost de la millor solucio fins ara
```

```
void recursiu (int v, int t, double c) {
    // v = darrer vertex del cami s
    // t = talla del cami s
    // c = cost del cami s
    if (t == n) {
        c += M[v][0];
        if (c < millor_cost) {
            millor_cost = c;
            sol = s;
            sol[v] = 0;
        }
    } else {
        for(int u = 0; u < n; ++u)
          if (u != v and s[u] == -1) {
            if (c + M[v][u] < millor_cost) {
                s[v] = u;
                recursiu(u, t+1, c + M[v][u]);
                s[v] = -1;
}   }   }   }
```

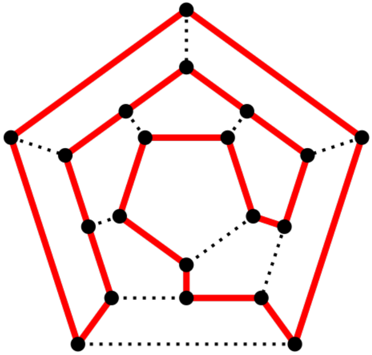
```
public:

    Viatjant (matriu M) {
        this->M = M;
        n = M.rows();
        s = vector<int>(n, -1);
        sol = vector<int>(n);
        millor_cost = infinit;
        recursiu(0, 1, 0);
    }

    vector<int> solucio (      ) { return sol;     }
    int         seguent (int x) { return sol[x]; }
    double      cost    (      ) { return millor_cost; }
};
```

Graf Hamiltonià

- Un cicle Hamiltonià és un cicle que visita cada vèrtex exactament un cop



Font: `https://en.wikipedia.org/wiki/Hamiltonian_path`

- Si un graf té un cicle Hamiltonià, llavors diem que el graf és Hamiltonià.
- Donat un graf, volem saber si és Hamiltonià.

Graf Hamiltonià

- Suposem que el graf és connex
- Suposem que el graf està representat amb llistes d'adjacència **ordenades**

```
typedef vector< vector<int> > Graf;
typedef list<int>::iterator iter;


class GrafHamiltonia {

    Graf G;                 // el graf
    int n;                  // nombre de vertexs

    vector<int> s;          // seguent de cada vertex
                            // (−1 si encara no utilitzat)

    bool trobat;            // indica si ja s'ha trobat un cicle
    vector<int> sol;        // solucio (si trobat)
```

Graf Hamiltonià

```
void recursiu (int v, int t) {
    // v = darrer vertex del cami, t = talla del cami
    if (t == n) {
        // cal assegurar-nos que el cicle es pugui tancar
        if (G[v][0] == 0) {
            s[v] = 0;
            trobat = true;
            sol = s;
            s[v] = -1;
        }
    } else {
        for (int u : G[v]) {
            if (s[u] == -1) {
                s[v] = u;
                recursiu(u, t+1);
                s[v] = -1;
                if (trobat) return;
}   }   }   }
```

Graf Hamiltonià

```
public:

    GrafHamiltonia (Graf G) {
        this->G = G;
        n = G.size();
        s = vector<int>(n, -1);
        trobat = false;
        recursiu(0,1);
    }

    bool te_solucio () {
        return trobat;
    }

    vector<int> solucio () {
        return sol;
    }
};
```