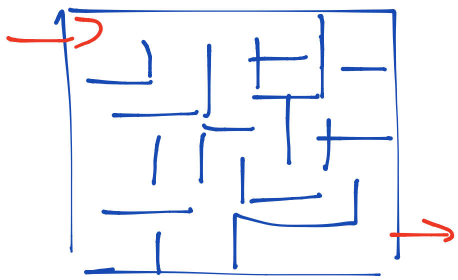
Grande famille d'algo. BACKTRACKING

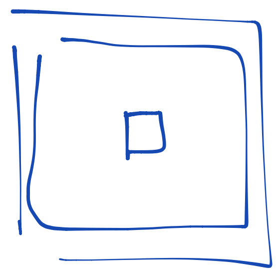
<u>idée</u> : recherche exhaustive dans un espace de recherche (souvent très grand).

- recherche systématique.
- exhaustive : on explore tout.
- eviter la redondance.

) chaque configuration sera examinée 1 et 1 seule fois.

Recherche d'un chemin dans un labyrinthe





Algorithme "brute force".

↳ on cherche partout.

<u>Schéma general</u>:

On va chercher des solutions de la forme $(a_1, a_2, \dots, a_n)$

↳ séquence de choix, une seq. de coups, ...

<u>Exemples</u>:

- <u>Sudoku</u>: un choix = 1 chiffre ds une case.
- lister tous les sous-ensemble d'un ens : $a_i = \top/\bot$ indiquant si l'élt i est dans le ss-ens. ou non.
- chemin dans 1 graphe : un "a_i" sera une transition/arc/arête.

A chaque étape de l'algo, on fait :

- on cherche comment compléter la solution partielle dont on dispose à l'instant.
- on les essaie <u>tous</u>. (ou jusqu'à trouver 1 solution).

Fondamentalement, on a un arbre que l'on va parcourir.

(a_1).

a_2^1 a_2^2 a_2^3 . . . a_2^ℓ — tous les choix possibles pour a_2

BT(S) : — une solution partielle = la config. courante de mon pb.

Si S est une solution à mon pb de départ → bingo!

Sinon :

- voir les possibilités $\mathcal{P}$ pour continuer.
- Pour tout $p \in \mathcal{P}$:
  - BT(S ∪ {p})

Lister tous les sous-ensembles d'un ens. de départ.

<u>input</u> : un tableau d'entiers. (tous différents).
↳ taille n, nom : E. indices 0… n−1

TLS (E, S, k) — S : la solution partielle en construction ; k → n° choix.

Si (k==n) Alors <u>afficher</u> le ss-ens. S → "complet"
↳ pour chaque élément de E, j'ai fait un choix.

<u>Sinon</u>:

- j'ajoute E[k] à S.
- TLS (E, S, k+1)
- j'enlève E[k] de S
- TLS (E, S, k+1)

Appel initial avec

TLS (E, ∅, 0)

Ex: le compte est bon.

jeu: choisir 6 cartes dans l'ens {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100} × 2
choisir un nb entre 100 et 999 → V

<u>obj:</u> faire une série de calculs à partir des cartes obtenues pour arriver à V.
opérations: +, −, ×, / .
↳ le reste doit être nul. x/y ssi x % y = 0
→ pas de nb < 0.

NB: - pas nécessaire d'utiliser toutes les cartes.
-

V = 243
L = [7, 8, 9, 3, 6, 6]

8+6 = 14	6/6 = 1	3×7 = 21
14−7 = 7	3×9 = 27	21+6 = 27
3×7 = 21	8+1 = 9	27×9 = 243
6+21 = 27	27×9 = 243	
27×9 = 243		

7, 8, 9, 3, 6, 6

7×8	7+8	8−7	8×9	8+9	9−8	
7×9	7+9	9−7	8×3	8+3	8−3	
7×3	7+3	7−3	8×6	8+6	8−6	
7×6	7+6	7−6	9×3	9+3	9−3	9/3
			9×6	9+6	9−6	
			3×6	3+6	6−3	6/3
			6×6	6+6	6/6	

$\boxed{\text{135 coups}}$

A chaque étape, on choisit 2 cartes, on les utilise pour 1 opération qui donne 1 nouvelle carte.

init. : 6 cartes
étape 1 : 5 cartes
étape 2 : 4
3 : 3
4 : 2
5 : 1

Au max 5 étapes.

1 configuration = un ens. de cartes.

1 coup : une op. à appliquer sur 2 cartes

1 solution : une séquence de coups.

CEB (L, V, Sol)
L → ens. de cartes
V → la valeur cible
Sol → la sol. en construction : la suite de coups qui a conduit à la config. L.

Si v ∈ L : | bingo!
| Afficher (Sol), return True

Sinon:

L_{cp} = la liste des coups possibles pour L, V

Pour chaque coup dans L_{cp} :

- Soit L_2 la liste L modifiée par coup.
- ajouter coup à Sol.
  L> empiler
- res = CEB(L_2, V, Sol)
- Si (res == True) : return True
- Retirer coup de Sol (Sol.pop())

Return False.

2 fonctions à définir.

→ L = [6, 5, 3, 4]
coup : (+, 3, 4)
→ L_2 = [6, 5, 7]

- liste des coups possibles pour L, V :

Pour chaque paire (x, y) d'élé^ts de L :
→ (+, x, y)
→ (*, x, y)
→ (/, x, y) si x % y == 0
→ (−, x, y) si x > y

+ éliminer les doublons.

- modifier L en fct d'un coup (op, x, y):

- retirer x, y de L
- ajouter ds L la valeur (x op y) → à calculer.

Exemple:

L = [5, 10, 2] V = 40

5+10 → [15, 2] → 15×2 = 30 ✗, 15+2 = 17 ✗, 15−2 = 13 ✗

5×10 → [50, 2] → 50×2 ✗, 50−2 ✗, 50+2 ✗, 50/2 ✗

10−5

10/5

10+2

10×2

10−2 → [8, 5] → 8+5 = 13 ✗, 8×5 = 40 !

10/2

5×2

5+2

5-2

8-5

Version itérative:

CEB (L, V)

Si $V \in L$: bingo! return True.

Sol = pile (vide) de coups

P = pile (vide) pour stocker des paires (liste de cartes , liste des coups restant à examiner)

Lcp = liste des coups possible pour L

P.empiler((L, Lcp))

tant que $P \neq \emptyset$:

- (L, Lcp) = P.pop()
- Si $Lcp \neq \emptyset$ Alors:
  - c = Lcp.pop()   : on pioche un coup c qui n'a pas encore été examiné.
  - P.empiler(L, Lcp)
  - L_2 = la liste L modifiée pour le coup c
  - Sol.empiler(c)
  - Si $V \in L_2$ Alors: bingo!
    - Afficher(Sol)
    - return True.
  - Sinon:
    - Lcp = liste des coups possible pour L_2
    - P.empiler((L_2, Lcp))
- Sinon: Si $Sol \neq \emptyset$ Alors Sol.pop()

retourner False.

Affichage des solutions :

input $\left[\begin{array}{l} Sol : [(op_1, x_1, y_1), (op_2, x_2, y_2), (op_3, x_3, y_3) \dots (op_k, x_k, y_k)] \\ L, V \end{array}\right.$

Ex: $Sol = [(+,2,2), (\times,5,6), (+,4,8), (+,30,7)]$

$L = [2,2,5,8,6,7]$

$V = 37$ $\quad \hookrightarrow \left[\begin{array}{l} 5\times 6 = 30 \\ 30+7 = 37 \end{array}\right.$

Algo: P = pile d'entiers

P.empiler(v). + S = l'ens. des op. pertinentes $= \emptyset$

tant que $P \neq \emptyset$:

- . val = P.pop()
- . soit (op, x, y) une op. dans Sol dont le res. est val.
- . enlever (op, x, y) de Sol.
- . S.empiler (op, x, y)
- . Si $x \in L$:
    - retirer une occurrence de x de L
    - Sinon P.empiler(x)
- . Si $y \in L$:
    - retirer une occ. de y dans L
    - Sinon P.empiler(y)

retourner S

Schéma algo itératif pour le backtracking.

BT():

P = pile (vide) de [config, liste de coups restant à explorer]

Soit c_0 la config. initiale

Soit L_{c_0} = la liste des actions possibles depuis c_0

P.empiler (c_0, L_{c_0})

tant que $P \neq \emptyset$:

    (c, Lc) := P.pop()

    Si $Lc \neq \emptyset$ Alors :

        coup = Lc.pop()

        P.empiler (c, Lc)

        n^elle config c' = c après modif de coup.

        Si c' est gagnant Alors $\boxed{\text{bingo!}}$

        Sinon :

            Lc' = liste des actions possibles depuis c'

            P.empiler (c', Lc')