

py sudoku.py > BT_sudoku >

```
1  DIM = 5
2  VALUES: list[int] = list(range(10))CH 3: BT
3
4  def coord(k: int, n: int) → tuple[int, int]:
5      """k: idx of current step, n:dimension of sudoku (i.e. 4x4 ⇒ n=4).
6      Take index in {0..(n-1)^2} and return index in {0..n}^2"""
7      return k // n, k % n
8
9  def discard_all(s: set, to_disc) → None:
10     for x in to_disc: s.discard(x)
11
12 def getCol(M: list[list], idx: int, k: int = 0) → list:
13     return [M[i][idx] for i in range(k)]
14
15 def T(x: list[list[int]], k: int, n: int = DIM) → set[int]:
16     """Explicit conditions, no 2 same nb on row or column"""
17     crt_val = set(VALUES)
18     i, j = coord(k, n) # idx of row, cols
19     discard_all(crt_val, x[i]) # discard already used values in row
20     if i > 0: # if (i, j) is not the first index of column j
21         discard_all(crt_val, [x[m][j] for m in range(i)]) #! i not j, discard already used values in column,
22         #* we process line by line so the amount of element filled in any column at step k is at most coord(k, n)[0]
23         # (i.e. row but works since k do not decrease)
24     return crt_val
25
26 def isSameDiag(i1: int, j1: int, i2: int, j2: int) → bool:
27     return abs(i1 - i2) == abs(j1 - j2) # i.e. |i1-i2| = |j1-j2|
28
29 def B(x: list[list[int]], k: int, n: int = DIM) → bool:
30     """Implicit conditions, no 2 same nb on the each diagonal"""
31     ki, kj = coord(k, n)
32     crt= x[ki][kj]
33     for m in range(k):
34         i,j = coord(m, n)
35         if isSameDiag(ki, kj, i,j) and crt == x[i][j]: return False
36     return True
37
38 def P(x: list[list[int]], k: int, n: int = DIM) → bool:
39     N: int = n**2
40     if k < N-1: return False #! if at least N-1 steps haven't been made ⇒ x never a solution
41     for crt_idx in range(N - 1):
42         if not B(x, crt_idx): return False
43     return True
44
45 def add_el(x: list[list[int]], yi: int, yj: int, y: int) → None:
46     """if row x[yi] not long enough ⇒ append element"""
47     if len(x[yi]) <= yj: x[yi].append(y)
48     else: x[yi][yj] = y
49
50 def BT_sudoku(dim: int) → list[list[int]]:
51     """dim: length of a row of sudoku grid"""
52     found: bool = False
53     def rBT(x: list[list[int]], k: int, n: int = DIM) → list[list[int]]:
54         """x nxn matrix initialized as n empty list, k in [l0, (n^2)-1 ] step idx"""
55         for y in T(x, k, n):
56             yi, yj = coord(k, n)
57             add_el(x, yi, yj, y)
58             if B(x, k, n):
59                 if P(x, k, n): return (x, found:=True)[0]
60                 rBT(x, k + 1, n)
61                 if found: return x
62     return rBT([[] for _ in range(dim)], 0, dim)
```

SUDOKU

tp3-nqueens-no-useless.py > solve_bt

```
1  def diff(arr1, arr2):
2      '''Return `arr1` \ `arr2` (mathematical difference)'''
3      set2 = set(arr2) # "is in" check should be O(1)
4      return [a1 for a1 in arr1 if a1 not in set2]
5
6  def isSameDiag(i1, j1, i2, j2):
7      '''isSameDiag(i1, j1, i2, j2) if the two points are on the same diagonal, and `False` otherwise.
8      Returns (variable) distj: Any inates are on the same diagonal (or anti-diagonal).'''
9      disti, distj = abs(i1-i2), abs(j1-j2)
10     return disti == distj
11
12 def T(x, k, n):
13     '''`T(x,k,n)` returns the set of all possible positions for the `(k+1)`-th queen, given the positions
14     of the first `k` queens
15     Parameters
16     -----
17     @ `x` - (Partial solution) i.e. Current state of the board. List of column indices
18     (i.e. [x_i]_{1 \leq i \leq k} where x_i = column of i-th queen)
19     @ `k` - the number of queens placed so far
20     @ `n` - the number of queens
21     Returns: The set of all possible positions for the `k+1`-th queen. '''
22     return diff(range(n), x[:k])
23
24 def B(x, k, n):
25     '''If the last queen is not on the same diagonal as any of the previous queens, then return True
26     Parameters
27     -----
28     @ `x` - (Partial solution) i.e. Current state of the board. A list of the queens column indices
29     (i.e. x0 x1 ... xk)
30     @ `k` - Index of current row/step we are working on
31     @ `n` - the size of the board
32     Returns: Whether the current solution is valid. i.e. If the last queen is not on the same diagonal
33     as any of the previous queens'''
34     if k == 0: return True
35     for i in range(k):
36         if isSameDiag(k, x[k], i, x[i]): return False
37     return True
38
39 def P(x, k, n):
40     if None in x: return False
41     for i in range(n-1):
42         crt = x[i]
43         for j in range(i+1, n):
44             if isSameDiag(i, crt, j, x[j]):
45                 return False
46     # checks each element shares a diagonals runs (n-1)*n/2 times
47
48     # Checking if 2 columns are the same
49     count = [0]*n
50     for el in x: # x are columns indices
51         count[el] += 1
52         if count[el] > 1: return False
53     return True
54
55 def solve_bt(n) \rightarrow None | list[list[int]]:
56     '''`solve_bt` returns all the solutions for the N-Queens problem for a chess board of size n.
57     Takes a number `n` and returns a (list of) lists of `n` columns indices, each of which is
58     such that the i-th queen is located at index `(i, solve_bt[k][i])` (for the k-th solution) of the
59     `n`x`n` chess board. No two queen are on the same column, same row or same diagonal.
60     Returns: A list of all possible solution to the N-Queens problem. i.e. list of lists of the columns indices of the queens.'''
61     if n < 4: return []
62     sols: set = set()
63     def bt_rec(x, k, n):
64         for y in T(x, k, n):
65             x[k] = y
66             if B(x, k, n):
67                 if P(x, k, n): sols.add(tuple(x)) # using a set to avoid duplicates
68                 bt_rec(x, k+1, n)
69
70     bt_rec([None]*n, 0, n)
71     return [list(sol_tuple) for sol_tuple in sols] # converting set of tuple back to list of list
```

N-QUEENS

Pour formaliser la méthode on considère un espace contenant des éléments \mathbf{x} de la forme d'un n -tuple

$$\mathbf{x} = (x_1, x_2, \dots, x_n)$$

avec

$$x_i \in S_i$$

où S_i est un ensemble fini qui contient les valeurs possible de x_i . Le problème à résoudre est spécifié par une propriété $P(\mathbf{x})$ à satisfaire. On cherche alors une solution ou toutes les solutions \mathbf{x} qui satisfont P .

Une façon de résoudre un tel problème est bien sûr d'essayer tous les \mathbf{x} possibles. Si $m_i = |S_i|$ est le nombre d'éléments dans S_i (le cardinal de S_i), alors il y a

$$N = m_1 \times m_2 \times \dots \times m_n = \prod_{i=1}^n m_i \quad \text{O(m^n) (m := |S|)} \\ \text{(complexité naïve)}$$

possibilités à considérer. C'est en général impraticable quand n et les m_i sont grands.

L'idée du backtracking est de n'explorer que les régions de S qui ont une chance de mener au succès. Pour cela, on construit la solution \mathbf{x} composante par composante, en considérant des solutions partielles

$$(x_1), \quad (x_1, x_2) \quad \dots (x_1, \dots, x_i) \dots (x_1, \dots, x_n)$$

Une solution partielle (x_1, \dots, x_i) est étendue à $(x_1, \dots, x_i, x_{i+1})$ avec

$$x_{i+1} \in T_{i+1}(x_1, x_2, \dots, x_i)$$

T_i défini quels sont les valeurs possibles / intéressantes à i . et ofc dépend des autres x_{i-k}

si cela ne contredit pas P . A priori, on pourrait penser que $T_{i+1} = S_{i+1}$, mais souvent, certaines valeurs de S_{i+1} peuvent facilement être exclues.

Pour savoir si une solution partielle contredit P , il faut pouvoir définir des fonctions booléennes

$$B_i(x_1, \dots, x_i), \quad i = 1 \dots n \quad B_n = P$$

$B_i = \text{False} \Leftrightarrow (x_1, \dots, x_i)$
ne pourra jamais faire partie d'une solution

qui sont fausses dans un tel cas. Ces fonctions sont appelées **fonctions d'arrêt** (ou *bounding function* en anglais). Elles permettent d'arrêter l'expansion d'une solution partielle qui n'a aucune chance de mener à une bonne réponse, et ainsi de diminuer la taille de l'espace exploré.

Dans la méthode du **backtracking**, si toutes les valeurs $x_{i+1} \in T_{i+1}$ conduisent à des valeurs fausses de B_{i+1} , on en déduit que le choix de x_i est à modifier.

On notera que la méthodes proposée ici de construire les solutions \mathbf{x} composante par composante produit une **représentation en arbre** de l'espace des solutions. En effet, les valeurs possible pour x_1 correspondent à plusieurs branches possibles à partir du début de la recherche, ou **racine** de l'arbre. A l'extrémité de chacune de ces branches partent d'autres branches qui représentent les valeurs de x_2 , et ainsi de suite jusqu'à x_n . Le backtracking explore cet arbre avec un **parcours en profondeur**, et avec l'espérance qu'il n'est souvent pas nécessaire d'explorer chaque branche jusqu'au bout.

Explore l'arbre en DFS, en changeant de branches si B_i est faux (contradiction).

Le nombre de branche à chaque niveau diminue !
(il est déterm par $|T_i(x_1, \dots, x_{i-1})|$)

=> on essaie la prochaine branche du MEME étage. Si aucune branche ne rend B_{i-1} vrai on remonte d'un étage.

On va ici donner une abstraction en python du processus de backtracking. Dans cet algorithme, on suppose ici que \mathbf{x} est une liste python de taille n , initialisée de façon arbitraire, par exemple

```
x=[None for i in range(n)]
```

On rappelle que les éléments de \mathbf{x} sont numérotés de 0 à $n - 1$, puisque c'est une liste python.

On suppose aussi que les fonctions d'arrêt sont spécifiées par une fonction **in** $\mathbf{B(x,k,n)}$ qui retourne vrai si (x_0, \dots, x_k) n'est pas impossible. On introduit aussi la fonction $\mathbf{T(x,k,n)}$ qui retourne une liste de valeurs possibles pour x_k , connaissant les valeurs de (x_0, \dots, x_{k-1}) . Cette liste est ici copiée dans $\mathbf{values[k]}$ afin de mémoriser les valeurs de x_k pas encore essayées.

Il faut initialiser \mathbf{values} comme une liste de $n + 1$ listes vides

```
values=[[] for k in range(n+1)]
```

On va ici jusqu'à $n + 1$ car, pour les besoins de la condition d'arrêt on demande que pour $k = n$, $\mathbf{T(x,k,n)}$ soit défini et retourne une liste vide.

Finalement, on définit aussi une fonction $\mathbf{P(x,k,n)}$ qui est vraie si (x_0, \dots, x_k) est une réponse au problème. Dans ce cas, on l'imprime grâce à la fonction $\mathbf{printSol(x,k,n)}$. En général, une solution qui vérifie P demande que $k = n - 1$. Mais, pour certains problème, un vecteur (x_0, \dots, x_k) partiel peut déjà satisfaire la propriété en question. On en verra un exemple plus loin, dans le problème de la somme d'un sous-ensemble de nombres.

Avec ces définitions, l'algorithme 10 décrit un code de backtracking qui donne toutes les solutions qui sont une réponse au problème.

Algorithm 10 Implémentation générique du *backtracking* avec la syntaxe python.

```

1 k=0
2 values [k]=T(x, k, n)
3 while k>-1:
4     if len ( values [k])==0:    on backtrack =>
5         k-=1
6     else:
7         x [k]=values [k].pop ()      (pop enlève bien de la liste)
8         if B(x, k, n):
9             if P(x, k, n): printSol(x, k, n)
10            k+=1
11        values [k]=T(x, k, n)    I. 10-11 : prochain
12                                            niveau de l'arbre,
13                                            prochaine sol

```

Il est important de comprendre plus en détail la façon dont le code 10 s'arrête. Une fois que tout l'arbre de solution a été exploré, le **backtracking** va remonter jusqu'à la racine. Comme toutes les valeurs de $\mathbf{values[0]}$ auront été épuisées, il y aura une étape de backtracking supplémentaire qui va faire passer k à -1. Cela va interrompre la boucle **while** à la ligne 3.

La version vue en cours nous donne que (donnée la taille du board n , nos conditions implicites B , conditions explicites/espace de solutions T et l'algorithme 11):

```
# Algorithm 11: Backtracing - recursive implementation
def rBacktrack (x, k, n) :
    for y in T(x, k, n) :
        x[k] = y
        if B(x, k, n) :
            if P(x, k, n) : print Sol(x, k, n)
            rBacktrack (x , k+1,n)
```

- On itère (récursevivement) sur les lignes du board en mettant à chaque fois 1 reine, ligne par ligne.
⇒ On a donc au maximum 1 reine par ligne.
- Les solutions sont donc les (indices des) colonnes du board,
i.e. une solution \mathbf{x} est un vecteur de $\llbracket 0, n-1 \rrbracket^n$, tel que la i^{eme} reine se trouve à l'index $(i, \mathbf{x}[i])$ du board. On a donc que le board est une matrice $M = (a_{ij}) \in \mathbb{M}_n$ telle que :

$$(a_{ij}) = \begin{cases} 1 & j = \mathbf{x}[i] \\ 0 & \text{sinon} \end{cases}$$

C'est ceci qui va donc nous définir nos contraintes **explicites**, celles exprimées par l'évolution de $T(x, k, n)$. Nos contraintes explicites vont être, ici, "Pas 2 reines sur la même colonne".

C'est à dire que à l'étape k , $T(x, k, n)$ va nous retourner l'espace des solutions que peut prendre x_{k+1} , i.e. "Quelles colonnes sont libres ?". On va simplement enlever les anciens x_i des prochains T l'ensemble $(\llbracket 0, n-1 \rrbracket \setminus \{x_0, \dots, x_k\})$

$T(x, k, n)$ se charge d'imposer les contraintes **explicites** (mises à jour à chaque k) et on itère dessus avec la boucle **for** de la ligne 3 (Algo 11)
⇒ On a donc maximum 1 reine par colonne.

- Les B sont les conditions **implicites**, celles qui vont vérifier que "pas 2 reines sont sur la même diagonale (ou anti-diagonale)". Dans les lignes 5 à 7 de l'algo 11, on ne passe au prochain k seulement si $B(x, k, n)$ return **True** i.e. seulement si les conditions implicites sur les diagonales sont respectés. C'est donc bien ce test qui va nous faire reculer si jamais on arrive à la fin de la boucle **for**.

⇒ On a donc maximum 1 reine par diagonale (ou anti-diagonale).

- $P(x, k, n)$ quant à elle, vérifie si jamais \mathbf{x} est une solution "finale" du problème, i.e. donne la condition d'arrêt (total) sur la solution courante. $P(x, k, n)$ détermine si on a fini de chercher ou non. C'est à dire l'ensemble des solutions S est défini comme $S := \{x | P(x, k, n)\}$.

CH. 2: GREEDY

```
coin(M: money, (c0, c1, ..., cn-1): coin set where c0 > c1 > ... > cn-1) =
    coin_rec(L, i, acc) =
        if L - ci >= 0
            then coin_rec(L - ci, i, acc + [ci])
        else if (i+1 < n && L > 0)
            coin_rec(L, i+1, acc)
    coin_rec(M, 0, NIL)
```

2.2 Pourquoi est-il greedy?

Car il essaie de toujours la meilleure option / le meilleur choix localement, pour espérer avoir le meilleur choix / meilleur solution globalement.

2.4 Optimalité

Cet algorithme est-il optimal ? Si oui prouvez-le, sinon donnez un contre-exemple.

Non il existe des valeurs de M et des coins sets pour lesquels l'algorithme ne donne pas la solution

2.4 Optimalité

Cet algorithme est-il optimal ? Si oui prouvez-le, sinon donnez un contre-exemple.

Non il existe des valeurs de M et des coins sets pour lesquels l'algorithme ne donne pas la solution optimale.

E.g. M (money) = 0.31, $C = \{25, 10, 1\}$

- Greedy : $0.31 = 1 \cdot 25 + 6 \cdot 1$ (7 pièces)
- Optimal: $0.31 = 3 \cdot 10 + 1 \cdot 1$ (4 pièces)

```

- def find_set(S, u):
    """":return: Index of 1st set in 'S' that contains 'u' """
-     for i in range(len(S)):
-         if u in S[i]: return i

- def union(S, Su, Sv, u_idx):
    """ Replace S[u_idx] by Su.union(Sv) and remove Sv from list of sets S """
    tmp = Su.union(Sv)
    S[u_idx] = tmp
    S.remove(Sv)

- def kruskal(A):
    """Given A, a square matrix (list of lists), returns a list of edges that compose the MST"""
    # NOTE: A weight of 0 means that there is no edge between nodes, and it should not be taken into the MST
    n = len(A)
    S, F, E = [{i} for i in range(n)], [], []
    for i in range(n):
        for j in range(i + 1):
            if A[i][j] != 0: E.append((i, j, A[i][j]))
    # visits each edge only once  $\Rightarrow O(|E|)$ 
    E = sorted(E, key=lambda edge: edge[2]) # sorted according to weight in increasing order
    for e in E:
        u, v = e[1], e[0]
        u_idx, v_idx = find_set(S, u), find_set(S, v)
        Su, Sv = S[u_idx], S[v_idx]
        if Su != Sv:
            F += [(u, v)]
            union(S, Su, Sv, u_idx)
    return F

```

(0-1 knapsack is just normal knapsack, not 0-1 means \Rightarrow allow repetition of item)

KRUSKAL

$\text{KRUSKAL}(V, E, w)$
 $A \leftarrow \emptyset$
for each vertex $v \in V$
 do $\text{MAKE-SET}(v)$
sort E into nondecreasing order by weight w
for each (u, v) taken from the sorted list
 do if $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
 then $A \leftarrow A \cup \{(u, v)\}$
 $\text{UNION}(u, v)$
return A

$\text{PRIM}(V, E, w, r)$
 $Q \leftarrow \emptyset$
for each $u \in V$
 do $\text{key}[u] \leftarrow \infty$
 $\pi[u] \leftarrow \text{NIL}$
 $\text{INSERT}(Q, u)$
 $\text{DECREASE-KEY}(Q, r, 0) \quad \triangleright \text{key}[r] \leftarrow 0$
while $Q \neq \emptyset$
 do $u \leftarrow \text{EXTRACT-MIN}(Q)$
 for each $v \in \text{Adj}[u]$
 do if $v \in Q$ and $w(u, v) < \text{key}[v]$
 then $\pi[v] \leftarrow u$
 $\text{DECREASE-KEY}(Q, v, w(u, v))$

The min cost arc from [nodes already in tree] to [nodes outside of tree] can be found by placing arc values in a priority queue.

Time Complexity :

- ▶ insert : $O(|V|)$
- ▶ Decrease-Key : $O(|E|)$
- ▶ Extract-Min : $O(|V|)$
- ▶ Using heap for priority queue : Each operation is $O(\log|V|)$
- ▶ Total : $O(|E| \log|V|)$

Ch1. Dvd & Conq:

Les données d'entrées sont supposées être dans une variable globale $A(1 : n)$ et on supposera aussi que le résultat du processus est stocké dans A . En d'autres termes, cela signifie que la solution du problème revient à modifier A . Ce sera le cas de plusieurs des exemples qu'on va considérer par la suite. Mais on peut facilement transformer ce pseudo-code pour faire apparaître explicitement une variable qui contient la réponse du problème, ou le modifier pour correspondre à un problème plus spécifique. On verra d'ailleurs plusieurs exemples concrets qui montrent comment ce pseudo-code peut s'adapter.

Le pseudo-code proposé ici utilise les fonction G , $SMALL$ et $COMBINE$ qu'on va décrire ci-dessous.

Le pseudo-code proposé ici utilise les fonction G , $SMALL$ et $COMBINE$ qu'on va décrire ci-dessous.

La fonction $SMALL(p, q)$ est un booléen qui détermine si le problème est assez petit pour utiliser une méthode de solution directe (ou naïve), implementée par la fonction G . La fonction $DIVIDE(p, q)$ retourne un entier m qui divise le problème $A(p : q)$ dans les sous problèmes $A(p : m)$ et $A(m + 1, q)$. Ces deux sous-problèmes sont résolus par des appels récursifs à $DivideConquer$ et sont combiné par une fonction $COMBINE$.

Algorithm 1 Abstraction de la méthode «diviser pour régner»

```
DivideConquer(int p, int q) # solve the problem on A(p:q)

    if SMALL(p,q) then G(p,q) # small problems are solved directly
    else
        int m=DIVIDE(p,q)      # propose a division of the problem
        COMBINE(DivideConquer(p,m), DivideConquer(m+1,q))
    endif
end DivideConquer
```

for each subproblem until trivial enough to use G

L'implémentation ci-dessus est récursive et c'est souvent ainsi que la méthode «Diviser pour régner» s'exprime naturellement. Mais on peut aussi donner une version itérative de la méthode. Un exemple sera proposé au paragraphe 1.5.

ou après une autre selon l'ordre alphabétique. On voit ici immédiatement que le programme ci-dessus nécessite $2(n - 1)$ comparaisons. On pose donc

$$T_{direct} = 2(n - 1) \quad (1.2)$$

On va voir ci-dessous que cette complexité n'est pas optimale. Mais avant cela, on va formuler le problème dans une optique «diviser pour régner».

Pour appliquer la méthode «diviser pour régner», il faut remarquer qu'on peut diviser A en deux parties, I_1 et I_2

$$I_1 = A[0 : \lfloor \frac{n}{2} \rfloor] \quad I_2 = A[\lfloor \frac{n}{2} \rfloor : n] \quad \text{[DIVIDE]}$$

Cela correspond à l'opération **DIVIDE** de l'algorithme **I**.

Si on obtient les **min** et **max** de I_1 et I_2 , on obtient facilement le **min** et le **max** de A

$$\begin{array}{ll} \text{min}(A) & = \text{Min}(\min(I_1), \min(I_2)) \\ \text{max}(A) & = \text{Max}(\max(I_1), \max(I_2)) \end{array} \quad \text{[Combine]} \quad (1.3)$$

C'est l'opération **COMBINE()** de l'algorithme **I**. Les fonctions **Min(a,b)** et **Max(a,b)** appliquées à deux nombres a et b sont les fonctions usuelles

```
def Min(a,b): Θ(1)           def Max(a,b): Θ(1)
    if a<=b: return a         if a>=b: return a
    else: return b            else: return b
```

Elles utilisent toutes deux une seule comparaison.

Le programme qui met en œuvre l'approche *diviser pour régner* pour trouver le **min** et le **max** d'une suite A est donné par l'algorithme **3**, en utilisant la syntaxe python.

Algorithm 3 Le code «diviser pour régner» pour rechercher le **min** et le **max**.

```
def minMax(A,i,j): i=begin, j=end indices
    if i==j:          # SMALL
        return (A[i], A[j])  ← O comp
    elif i==j-1:       # SMALL
        if A[i]<A[j]:    ← SMALL
            return (A[i], A[j])  ← 1 comp
        else:
            return (A[j], A[i])  ← (pair de)
    else:
        mid=int( (i+j)/2 ) # DIVIDE
        (fmin,fmax)=minMax(A, i, mid)  ← Min, max de grp1
        (hmin,hmax)=minMax(A, mid+1, j) → Min, max de grp2
        return ( min(fmin,hmin), max(fmax,hmax) ) # COMBINE
```

Algorithm 5 Une implémentation récursive du quicksort

```
from partition import partition

def quickSort(A,p,q):
    if p>=q: return # end condition
    j=partition(A,p,q)
    quickSort(A,p,j-1)
    quickSort(A, j+1,q)

# --- example ---
A=[6,1,7,-5,2,8,4]
quickSort(A,0,len(A)-1)
print A  # produces [-5, 1, 2, 4, 6, 7, 8]
```

section qu'on souhaite trier. Cette fonction modifie l'ordre des éléments dans A . De plus elle retourne un indice q , qui correspond à la position du pivot choisi dans la suite A réorganisée, de sorte que

section qu'on souhaite trier. Cette fonction modifie l'ordre des éléments dans A . De plus elle retourne un indice q , qui correspond à la position du pivot choisi dans la suite A réorganisée, de sorte que

$$A[m : q] \leq A[q] \leq A[q + 1 : p + 1]$$

où l'on a utilisé la syntaxe python avec laquelle $A[m : q]$ contient les éléments d'indices m à $q - 1$.

Le principe de cet algorithme est le suivant. La section de la suite à diviser est parcourue du début vers la fin avec l'indice i et de la fin vers le début avec l'indice q . Le parcours montant s'arrête si on trouve un élément $A[i]$ supérieur ou égal au pivot, et le parcours descendant s'arrête dès qu'on trouve un élément

Algorithm 6 Implémentation de la fonction `partition`

```

1 def partition (A,m,p):
2     if m==p: return m
3     pivot=A[m]
4     i=m
5     q=p+1
6     while i<q:
7         while 1:
8             i+=1
9             if i>=p or A[ i]>=pivot: break
10        while 1:
11            q-=1
12            if A[ q]<= pivot: break
13        if i<q: (A[ i] ,A[ q])=(A[ q] ,A[ i])
14        (A[m] ,A[ q])=(A[ q] ,A[m])
15    return q

```

$A[q]$ inférieur ou égal au pivot. Ce sont les lignes 7-9 et 10-12 de l'algorithme 6 qui implémentent cette partie. Une fois ces deux éléments trouvés, on échange leurs positions (ligne 13). Cela signifie alors que tous les éléments de A entre m et i vérifient la propriété d'être inférieurs ou égaux au pivot, alors que tous les éléments entre q et p sont garantis d'être supérieurs ou égaux au pivot. On notera que le parcours montant commence avec $i = m + 1$, car le pivot $A[m]$ est déjà correctement placé dans la zone des indices inférieurs à i .

[More implementation details on how to handle with algorithm 6](#)

Algorithm 7 Implémentation itérative du quicksort

```

1 def quickSort (A,p,q):
2     stack = []
3     while 1:
4         while p<q:
5             j=partition (A,p,q)
6             if j-p < q-j : # store the longest section
7                 stack.append(j+1); stack.append(q)
8                 q=j-1
9             else:
10                 stack.append(p); stack.append(j-1)
11                 p=j+1
12             if len(stack)==0: return
13             q=stack.pop(); p=stack.pop()
14
15 A=[3,1,5,6,2,1,8,3]
16 quickSort(A,0,len(A)-1)
17 print A
def is_majority(A, element):
    """Tells if 'element' is the majority element in A"""
    if element is None or A is None: return False
    n = len(A)
    if n == 1: return element == A[0]
    anc = n // 2 + 1 # * anc := appearance number constraint. number of times it must appear to be the majority element

```

```

def is_majority(A, element):
    """Tells if 'element' is the majority element in A"""
    if element is None or A is None: return False
    n = len(A)
    if n == 1: return element == A[0]
    anc = n // 2 + 1 # * anc := appearance number constraint. number of times it must appear to be the majority element
    i, crt_an = 0, 0 # * current appearance number
    for el in A:
        if el == element:
            crt_an += 1
            if crt_an >= anc: return True
    else: return False

def reduce(A):
    """Reduces A in at most len(A) // 2 parts using pair-wise votes"""
    n = len(A)
    if n <= 1: return A # * If A is empty or a singleton  $\Rightarrow$  result will be A itself
    A_prime = []
    for i in range(0, n - 1, 2):
        a, b = A[i], A[i + 1]
        if a == b: A_prime.append(a)
    return A_prime

def dandc(A):
    """Divide and Conquer algorithm"""
    n = len(A)
    if n == 0: return None
    if n == 1: return A[0]

    def dandc_rec(A_prime):
        l = len(A_prime)
        if l == 1: return A_prime[0]
        if l == 0: return None
        if l % 2 == 1:
            rand_el = A[-1]
            if is_majority(A, rand_el): return rand_el
            else: A.remove(rand_el) # Now l is even
        return dandc_rec(reduce(A_prime))

    candidate = dandc_rec(A)
    return candidate if is_majority(A, candidate) else None

def exp_dandc(base, p):
    if p == 0: return 1
    if p == 1: return base
    from math import log2
    """Divide and Conquer implementation of exponentiation"""
    def toBin(x):
        return bin(x)[2:][::-1]

    binP, i0, stop = toBin(p), 1, int(log2(p))

    def exp_rec(prod, crt_pow, i):
        """ sum : contains the sum of the base^(2^i)
        crt_pow: the current power of k (should be equal to k^(2^i))
        binP: contains the binary representation of k (as an array)
        i : the current step
        stop: the max number of step i.e. stopping condition is "i >= stop" (stop := floor(log2(p)))
        """
        if i > stop: return prod
        crt_pow = crt_pow * crt_pow
        if binP[i] == '1': prod *= crt_pow
        return exp_rec(prod, crt_pow, i + 1)
    prod0, pow_0 = 1, base
    if binP[0] == '1': prod0 = base
    # if P is odd then we have to multiply by base at the beginning  $\Rightarrow$  hence why we start our product at base
    return exp_rec(prod0, pow_0, i0)

```

MAJORITY_EL

EXP D&C

Ch4. PROG DYN:

1.1 Fonction de récurrence

- Soit n le nombre de ville et soit $C(i, j)$ la fonction qui retourne tous les chemins possible pour aller de la ville i à la ville j . Soit donc $C'(i, j) := \min(C(i, j))$ la fonction qui retourne le coût optimal (minimal) pour aller de la ville i à la ville j .

Le but de cet exercice est donc de trouver un algorithme qui déterminera $C'(i, j)$ pour tous $i, j \in \{0 \dots n - 1\}$.

Donnée la matrice M (triangulaire inférieure) des coûts initiaux dans l'énoncé, on cherche à remplir la matrice $T = [T_{ij}]_{1 \leq i, j \leq n} \in \mathbb{M}_n$ des coûts optimaux où on définit T_{ij} comme suit:

$$T_{ij} := \begin{cases} \text{None} & \text{if } j > i \quad (\text{car triangulaire inférieure}) \\ C'(i, j) & \text{if } j < i \\ 0 & \text{if } i = j \end{cases}$$

Notre relation de récurrence est donc :

$$(T_k)_{ij} = \min_{j < k < i} ((T_{k-1})_{ij}, (T_{k-1})_{kj} + (T_{k-1})_{ik})$$

(j = départ, i = arrivée). Soit v_i la ville n° i et $p(i, j)$ le chemin de v_i à v_j . Cette relation de récurrence de T tient du fait que si p est le trajet optimal pour aller de v_i à v_j (i.e. celui dont le coût vaut $C'(i, j)$) alors pour toute étape k et pour tout i, j :

Soit $v_k \in p$ et dans ce cas $p(k, j)$ et $p(i, k)$ sont de coût optimaux
i.e. $C'(i, j) = C'(k, j) + C'(i, k) \implies (T_k)_{ij} = C'(k, j) + C'(i, k) = (T_{k-1})_{jk} + (T_{k-1})_{ik}$

Soit $v_k \notin p$ et dans ce cas $(T_{k-1})_{ij}$ ne change pas, i.e. $(T_k)_{ij} = (T_{k-1})_{ij} = C'(i, j)$

Si ce n'est pas le cas, alors cela implique qu'il existe un autre trajet de coût inférieur aux 2 options plus haut ce qui contredit le fait que p soit le trajet optimal i.e. que le coût de p soit $C'(i, j)$.

Cette relation s'exprime donc le fait que à chaque étape k on prend le minimum entre $(T_{k-1})_{ij}$ et $(T_{k-1})_{jk} + (T_{k-1})_{ik}$.

- La formule a-t-elle un impacte sur la manière de remplir T ?

La formule à bien sûr un impact car c'est cette relation de récurrence qui vient définir. En effet T est remplie selon la définition "piecewise" plus haut, et chaque $C'(i, j)$ est donc donné par la relation de récurrence.

Il faut de plus s'assurer que les valeurs utilisés soient bien "initialisés" au moment où l'on est entrain de les remplir. Par exemple on à la relation que i doit toujours être $> j$, ou encore que $k > j$, et c'est bien la relation de récurrence qui nous donne cette structure / manière de remplir T .

1.3 Complexité en temps

- La complexité en temps est de $O(n^3)$, en effet on doit optimiser 3 paramètres en fonction l'une de l'autre. C'est à dire, on doit optimiser le paramètre "coût" en fonction j par rapport i et k , puis en fonction de i par rapport à j et k , puis finalement en fonction de k par rapport à i et j , on a donc 3 "depths" / niveau de profondeur.

On aurait donc 3 boucles `for` où l'on calcule l'opération `min(a, b)` où a et b on été défini avant, ce sont des scalaires (i.e. pas des listes) `min` prend donc un temps constant (1 opération: `a > b ?`). Elle est répété au plus n^3 fois ce qui fait bien $O(n^3)$.

On a bien 3 boucles `for`, celles sur j et i sont en $O(n^2)$ fois. En effet elles sont bornées par une boucle imbriqué de ce type

```
for i in range(n):
    for j in range(i):
        #<something in O(1)>
```

qui tournerait $\sum_{j=0}^n j = \frac{n \cdot (n+1)}{2}$ fois, et $n \cdot (n+1) = n^2 + nO(n^2)$ donc en rajoutant la boucle sur k qui fait n iterations et l'opération `min` qui est en $O(1)$ on a bien $O(n^2) \cdot O(n) \cdot O(1) = O(n^3)$

```
def getpath(i, j, P):
    """ return optimal path with the optimal intermediary steps in P"""
    intermed = sorted(P[i][j])
    opti_path = [i]
    for inter_step in intermed:
        opti_path += [inter_step]
    opti_path += [j]
    return opti_path

def get_solution(M):
    """Returns a table T where each value T_ij tells the minimum cost to get from city i to city j based on the cost matrix M. Also returns the cost of travelling from city 0 to n-1 and the respective path as a list of indexes."""
    # We start by setting T at M, like in Floyd algorithm, then
    # we will use those values to compute some intermediary values (optimal solutions for the subproblems)
    # and then by just continuing the algorithm and using our previous we will finally get the optimal solution in Tij for all ij
    T, n = deepcopy(M), len(M) # M is a squared lower triangular matrix.
    P = [[[] for _ in range(n)] for _ in range(n)]
    # matrix that will contain the optimal intermediary steps to take to minimize the cost.
    # I.e. (when filled) P(i,j) is the path to take to pay the min. amount Tij (to go from j to i).
    # (since T is lower triangular we must always have j <= i)
    for k in range(n):
        for i in range(k+1, n):
            for j in range(min(k+1, i-1)):
                if j == i: continue
                # T triangular => we then iterate only over the necessary values of k,i,j
                # that is j < k < i.
                # We could add i=k and k=j but Tij = 0 for all i=j so we dont need to cover them
                # since they'll be 0.
                crt, candidate = T[i][j], T[i][k] + T[k][j]
                if candidate < crt : # if T_k-1(i, k)+ T_k-1(k, j) < T_k-1(i, j) we set T_k(i, j) to T_k-1(i, k)+ T_k-1(k, j)
                    # or else we dont change it
                T[i][j] = candidate
                P[i][j].append(k) # we add this step to the matrix of paths to keep a record of the optimal intermediary
                # step to take
    # Tij now contains the contains min(C(i, j)) (for i,j in [0, n-1]) where C(i, j) would be defined
    # as the function that returns all possible path to go from i-th town to j-th town.
    opti_path = getpath(n-1, 0, P)
    return T, T[n-1][0], opti_path

def compute_change(money, coin_set:list) -> list:
    """Computes the optimal change (lowest number of coins) for the given 'money' and 'coin set'.
    Parameters
    -----
    @ `money` - amount to change
    @ `coin_set` - values of the coins to used when returning the money (sorted in decreasing order)
    Returns: (list) Solution ``l`` s.t. ``l[i]`` is the amount of times ``coin_set[i]`` was used."""
    if coin_set is None or coin_set == [] or money <= 0: return []
    C, n = coin_set, len(coin_set)
    # matrix of solutions where the row i contains the solution for the subproblem with C = Ci = {cn, ..., ci}. (we start with the lowest coin values)
    # say in rapport maybe that e.g. for C = {10,5,1} we start with C={1} then C={1,2}
    D = []
    # Let S be the optimal answer
    # Either there exists a coin of value exactly 'money' (i.e. S = <that coin>)
    for i in range(n):
        coin = C[i]
        if money == coin:
            sol = [0]*n
            sol[i] = 1
            return sol
    # if not then there exists at least 2 subsets S1, S2 such that S1, S2 also minimize the amount of coin used (if S is optimal)
    # (S1, S2) are the solutions to the subproblems, so we can compute the subsolutions step by step by memorizing the answers to the previous steps in D
    def solve_sub(Ci, step_idx):
        """ Returns the list res of coinset where res[k] is the amount of times we used coin Ci[k]."""
        left, sol = money, [0]*n
        for k, coin in enumerate(Ci):
            # if largest current coin is too big, then the solution is exactly the one we memorized before
            if (k == 0 and step_idx > 0 and coin > left): return D[step_idx-1]
            crt_amnt, left = (left // coin), (left % coin)
            sol[n-step_idx-1 + k] = crt_amnt # stores crt_amnt in sol but in reversed order since we iterate first on the lowest coins
            if left <= 0: return sol
        return sol
    D = [None]*n
    D[0] = [1]
    for i in range(1, n):
        D[i] = solve_sub(C[i:], i)
    return D[n-1]
```

TRAIN

COIN DYN

```

if k == v and step_idx > 0 and coin > left: return v[step_idx-1]
crt_amnt, left = (left // coin), (left % coin)
sol[n-step_idx-1 + k] = crt_amnt # stores crt_amnt in sol but in reversed order since we iterate first on the lowest coins
if left <= 0: return sol
return []

for i in range(n-1, -1, -1):
    tmp = solve_sub(C[i:], n-1-i) # padding with the necessary amount of 0s
    D.append(tmp)

# Now we just have to search the min of the sum of each column
# Hence min { sum(D[i]) } for i in {0..n} is the optimal amount of coins, and row i contains the optimal solution.
opti_row = min(range(n), key= lambda i: sum(D[i]))
out = D[opti_row]
return out

```

A dynamic programming solution

What would be a *Dynamic Programming* approach instead ?

Recall Starting point :

- ▶ We look at a problem that can be split into **subproblems**.
- ▶ Subproblems have the **same structure** as the overall problem.
- ▶ The **same subproblem** occurs **several times**.
- ▶ We want to find an **optimal** solution **efficiently**.

Intuition of Dynamic Programming :

1. We first solve the “smallest” **subproblems optimally** and **gradually build the solution** of the overall problem out of them in a **specific order**.
2. Values for subproblems on the way to the overall solution are **never recalculated** and thus we save time.

A dynamic programming solution

Idea : Solve first for one cent, then two cents, then three cents, etc., up to the desired amount. Save each answer in an array.

For each new amount N , compute all the possible pairs of previous answers which sum to N .

That's a bottom-up approach !

For example, to find the solution for $13c$:

- ▶ **First**, solve for all of $1c$, $2c$, $3c$, ..., $12c$
- ▶ **Next**, choose the best solution among :
 - ▶ Solution for $1c$ + solution for $12c$
 - ▶ Solution for $2c$ + solution for $11c$
 - ▶ Solution for $3c$ + solution for $10c$
 - ▶ Solution for $4c$ + solution for $9c$
 - ▶ Solution for $5c$ + solution for $8c$
 - ▶ Solution for $6c$ + solution for $7c$

Going into further detail ...

Suppose coins are 1c, 3c, and 4c

- ▶ There's only one way to make 1c (one coin)
- ▶ To make 2c, try $1c+1c$ (one coin + one coin = 2 coins)
- ▶ To make 3c, just use the 3c coin (one coin)
- ▶ To make 4c, just use the 4c coin (one coin)
- ▶ To make 5c, try
 - ▶ $1c + 4c$ (1 coin + 1 coin = 2 coins)
 - ▶ $2c + 3c$ (2 coins + 1 coin = 3 coins)
 - ▶ The first solution is better, so best solution is 2 coins
- ▶ To make 6c, try
 - ▶ $1c + 5c$ (1 coin + 2 coins = 3 coins)
 - ▶ $2c + 4c$ (2 coins + 1 coin = 3 coins)
 - ▶ $3c + 3c$ (1 coin + 1 coin = 2 coins) – best solution
- ▶ Etc.

We presented a Divide and Conquer algorithm that is recursive with a branching factor of 5.

- ▶ The algorithm takes exponential time with base 5.

While our dynamic programming algorithm is $O(N*K)$, where N is the desired amount and K is the number of different kinds of coins.

Two key ingredients an optimization problem must have in order for dynamic programming to apply :

1. optimal substructure and
2. overlapping subproblems

1. Optimal substructure

- ▶ Dynamic programming is a technique for finding an optimal solution.
- ▶ The **principle of optimality** describes the property of having an optimal substructure.
- ▶ The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems.

Example : COINS - The problem of making amount N_c with the fewest number of coins.

- ▶ Either there is an N_c coin, or
- ▶ The set of coins making up an optimal solution for N_c can be divided into two nonempty subsets, N_{1c} and N_{2c} .
- ▶ If either subset, N_{1c} or N_{2c} , can be made with fewer coins, then clearly N_c can be made with fewer coins, hence solution was **not** optimal.

Beware when applying the principle :

- ▶ The principle of optimality holds if
 - ▶ Every optimal solution to a problem contains...
 - ▶ ...optimal solutions to all subproblems
- ▶ The principle of optimality does **not** say
 - ▶ If you have optimal solutions to all subproblems...
 - ▶ ...then you can combine them to get an optimal solution

Example :

- ▶ The optimal solution to $7c$ is $5c + 1c + 1c$, and
- ▶ The optimal solution to $6c$ is $5c + 1c$, but
- ▶ The optimal solution to $13c$ is **not** $5c + 1c + 1c + 5c + 1c$
- ▶ But there is some way of dividing up $13c$ into subsets with optimal solutions (say, $11c + 2c$) that will give an optimal solution for $13c$.

Hence, the principle of optimality holds for this problem.

Example 4 : All-pairs shortest path problem

Problem : Given a directed weighted graph (without negative cycles), find the shortest path between every node-pair in the graph.

- ▶ $G = (V, E)$ where nodes in V are numbered $1, 2, \dots, n = |V|$
- ▶ Matrix L with $L(i, j) =$ the length of edge (i, j) and
 - ▶ 0 if $i = j$
 - ▶ ∞ if the edge does not exist
- ▶ Output matrix $D(i, j)$ with the distances

Note : There are other famous algorithms for the problem version of finding the shortest path between just two nodes (*Dijkstra* for non-neg. weights and *Bellman-Ford* if there are neg. weights, but no neg. cycles).

1. Optimal substructure property holds :

If k is a node on the shortest path from i to j then the paths :

- ▶ from i to k must be optimal
- ▶ from k to j must be optimal

2. Recursive solution that 3. exploits the overlapping substructure :

- ▶ Let D be a matrix with the length of the shortest paths
- ▶ Initially $D=L$
- ▶ After k interactions D gives the length of the shortest paths that uses only nodes from 1 to k
- ▶ After n interactions D gives the solution
- ▶ At the k th interaction there are two options for the path between i and j :
 - ▶ the path does not pass by the node k and then : $D(i, j)$ does not change between the two interactions
 - ▶ otherwise $D(i, j) = D(i, k) + D(k, j)$
- ▶ Therefore, $D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$
- ▶ The algorithm finishes when $k=n$.

Floyd's algorithm

```
function Floyd(L[1..n,1..n])
```

- ▶ $D = L$
- ▶ for $k = 1$ to n do
 - ▶ for $i = 1$ to n do
 - ▶ for $j = 1$ to n do
 - $D[i,j] = \min(D[i,j], D[i,k] + D[k,j])$
- ▶ return D .

Complexity : $O(n^3)$

- ▶ To get the paths we can use a second matrix for the nodes k used in the path.
- ▶ Given two sequences $x[1..m]$ and $y[1..n]$, find the longest subsequence which occurs in both.
- ▶ **Example :** $x = A B C B D A B$, $y = B D C A B A$
- ▶ B C and A A are both subsequences of both
- ▶ What is a LCS ?

Dynamic Programming Approach :

- ▶ Define $c[i,j]$ to be the length of a LCS of subsequences $x[1..i]$ and $y[1..j]$
- ▶ Initial cases : $c[0,j] = c[i,0] = 0$
- ▶ Recursive formula for LCS :
 - ▶ $c[i,j] = c[i-1,j-1] + 1$ if $x[i] = y[j]$
 - ▶ $\max c[i,j-1], c[i-1,j]$ otherwise
- ▶ Final answer $c[m,n]$ (computed bottom-up)
**See reference book *Introduction to Algorithms* by Cormen et al., page 394 for complete algorithm - accessible online.*

Running time : $O(mn)$.

Ch5. BRANCH BOUND:

- (a) Quel est le rôle de $h(x)$ dans la fonction de coût $\hat{c}(x) = g(x) + h(x)$?
- (b) A quel type de recherche correspond la fonction de coût
- 1) $\hat{c}(x) = g(x) + h(x)$?
 - 2) $\hat{c}(x) = h(x)$?
- (c) En se basant sur les critères discutés en cours concernant les propriétés de \hat{c} vis-à-vis d'une fonction de coût idéale $c(x)$, montrez que si l'on fait une recherche Branch-and-Bound avec la fonction de coût définie au point précédent et que l'on trouve une solution, cette solution est optimale.

- (a) Avant de parler du rôle de la fonction $h(x)$, il faut d'abord expliquer pourquoi il est avantageux (dans ce contexte) de garder notre $\hat{c}(x_j)$ le plus constant possible pour tout j .

Soit T , l'arbre des possibilités de racine x_0 (parfois appelé *root*) du jeu (où une solution de notre problème est simplement une suite de edge de T) et soit x_j un noeud de T , i.e. x_j est un état de notre problème.

On veut garder $\hat{c}(x) \leq \hat{c}(y)$ pour tout enfants y de x , car pour une recherche directe qui minimise le coût $\hat{c}(x)$, le relation (3.3) du cours donne que $c(x) \leq c(y)$.

Or \forall enfant y de x : $c(x) \leq c(y)$ implique que $\hat{c}(x^*) = c(x^*) \geq c(x_0) = c(optimum)$. (x^* est un noeud solution)

Donc vu que le coût des noeuds dans la descendance d'un x_0 et jusqu'à x^* ne peut pas diminuer, et que x^* est garantie d'être une solution optimale, le mieux qu'on puisse faire (pour minimiser le coût du chemin total $x_0 \dots x_i \dots x^*$) est de ne pas augmenter le coût des $x_i > 0$ i.e. le garder constant.

En résumé, on veut garder les $\hat{c}(x_{i+k})$ le plus proche de $c(x_0)$ et donc des autres $\hat{c}(x_i)$ possible. Ce qui ce traduit par "garder le coût" constant car la relation (3.3) nous garanti que $c(x_0)$ est le plus petit coût possible. (i.e. le coût de la solution qui minimise le plus le coût total)

(Ce qui est particulièrement dur quand \hat{c} est une mauvaise approximation i.e. les $\hat{c}(x)$ sont loin des $c(x)$)

$c(x_0) \approx \hat{c}(x_0) \approx \hat{c}(x_i) \approx \hat{c}(x_{i+k}) \approx \hat{c}(x^*)$ serait une recherche parfaite (pour un $\hat{c}(x_0)$ proche de $c(x_0)$, $k, i \in \mathbb{N}$)

On peut donc maintenant constater que le rôle de h est d'imposer que le coût des x_{i+k} ne peut rester constant que si les $g(x_{i+k})$ diminuent. C'est à dire que si g est une "mauvaise" borne/fonction (g est une borne inférieure du nombre de coup qui reste à faire), cela va directement se voir sur le coût total \hat{c} et les $\hat{c}(x_{i+k})$ (pour les plus hautes valeurs de k). En effet on a que:

$$\begin{aligned} \hat{c}(x_{i+k}) \leq \hat{c}(x_i) &\iff g(x_{i+k}) + h(x_{i+k}) \leq g(x_i) + h(x_i) \\ &\iff g(x_{i+k}) + h(x_i) + k \leq g(x_i) + h(x_i) \\ &\iff g(x_{i+k}) + k \leq g(x_i) \\ &\iff k \leq g(x_i) - g(x_{i+k}) \end{aligned}$$

Ce qui veut dire que pour garder le coût constant, il faut que entre $g(x_i)$ et $g(x_{i+k})$ on se soit rapproché de la solution optimale x^* de k coups.

Or comme on fait seulement un mouvement par étape, (i.e. $\max(g(x_i) - g(x_{i+1})) = 1$) cela implique qu'à chaque étape on se rapproche de x^* autant que possible, i.e. qu'on prenne le meilleur chemin / E-Node à chaque étape.

Ce qui nous donne la relation "on est sur le bon chemin" ($\hat{c}(x_{i+k})$ constants) si et seulement si on le nombre de case mal placé diminue (i.e. $g(x_{i+k}) < g(x_i)$).

- (b) A quel type de recherche correspond la fonction de coût

- 1) $\hat{c}(x) = g(x) + h(x)$?

C'est une recherche "least cost" (comme défini en (3.4) dans le cours), on est dans le cas d'un jeu où l'espace à explorer est un graph dirigé qui décrit les différents mouvements possibles à partir d'une configuration x_0 .

On cherche donc une configuration gagnante x^* telle que la longueur du chemin $path(x_0, x^*)$

jeu ou l'espace à explorer est un graphe dirigé qui décrit les différents mouvements possibles à partir d'une configuration x_0 .

On cherche donc une configuration gagnante x^* telle que la longueur du chemin $path(x_0, x^*)$ i.e. la somme des coûts des noeuds du chemin $(x_0 \dots x_i \dots x^*)$ soit minimale.

Pour ce faire on définit donc $h(x)$ comme étant la profondeur de la configuration x dans l'arbre de recherche, (*height*) (i.e. le nombre de coups déjà fait / coût accumulé jusqu'à x) et $g(x)$ comme une lower bound du nombre de coups restant à faire pour atteindre x^* .

Comme ça, on a une estimation du nombre de coups minimum qui compose $path(x_0, x^*)$. (un coup == un edge de l'arbre)

2) $\hat{c}(x) = h(x)$?

D'après la relation (3.4) du cours, encore une fois, on aurait à faire à une recherche à *coût uniforme*.

“On peut bien sûr choisir $g(x) = 0$ pour tout x . On parle de recherche à coût uniforme.

Le Branch & Bound trouvera l'optimum recherché, mais l'efficacité de l'exploration sera sans doute faible.”

On remarque que dans ce cas la recherche “least cost” nous donne en fait un BFS (breadth first search).

car seul la profondeur détermine le coût donc on va explorer les noeuds sur la même largeur de x en priorité. Elle est donc très peu efficace, niveau complexité en temps car il n'y aucun “guide” sur le chemin à prendre.

3) Les critères et propriétés de $\hat{c}(x)$ ayant été longuement abordés au point 1) on se contentera de rappeler le théorème 2 du cours:

“Soit une recherche Branche & Bound à coût minimum (least cost) et soit une fonction \hat{c} qui vérifie que $\hat{c}(x) \leq c(x)$ pour tous les noeuds x , et $\hat{c}(x^) = c(x^*)$ pour les noeuds réponse x^* .”*

Donc en se basant sur le fait que l'énoncé $\hat{c}(x) = g(x) + h(x)$ vérifie bien les critères du théorème 2, il ne nous reste plus qu'à prouver que $h(x) \leq g(x) + h(x)$ (car $g(x) + h(x) \leq c(x)$) et que $h(x^*) = h(x^*) + g(x^*)$ (car $g(x^*) + h(x^*) = c(x^*)$):

$$\begin{aligned} h(x) &\leq g(x) + h(x) \\ \iff h(x) - h(x) &\leq g(x) \\ \iff 0 &\leq g(x) \end{aligned}$$

Or $g(x)$ est une lower bound du nombre de coups restant à faire pour atteindre x^* elle donc évidemment toujours plus grande à 0. Sauf pour $g(x^*)$ qui vaut 0.

$$\begin{aligned} h(x^*) &= h(x^*) + g(x^*) \\ \iff h(x^*) &= h(x^*) + 0 \\ \iff h(x^*) &= h(x^*) \end{aligned}$$

Donc par le théorème 2, si on trouve une solution avec cette fonction de coût ($\hat{c}(x) = h(x)$) elle est optimale.

3 Plus court chemin (Extra 1 Point)

3.1 Fonction de coût

La fonction de coût qui remplit les critères nécessaires pour garantir l'optimisation d'une solution est tout simplement la même que celle utilisée pour le problème du taquin.

En effet le théorème 2 du cours et le “guide” sur comment choisir sa fonction de coût sont tellement généraux qu'on pourrait les appliquer à beaucoup de problèmes. La seule chose qui va changer ici, fondamentalement, c'est la fonction $g(x)$ qui a été choisie pour être simplement la distance définie par la norme L_1 (i.e. $\|x - y\|_1$) entre la position de x et la destination y .

3 Plus court chemin (Extra 1 Point)

3.1 Fonction de coût

La fonction de coût qui remplit les critères nécessaires pour garantir l'optimisation d'une solution est tout simplement la même que celle utilisée pour le problème du taquin.

En effet le théorème 2 du cours et le "guide" sur comment choisir sa fonction de coût sont tellement généraux qu'on pourrait les appliquer à beaucoup de problèmes. La seule chose qui va changer ici, fondamentalement, c'est la fonction $g(x)$ qui a été choisie pour être simplement la distance définie par la norme L_1 (i.e. $\|\cdot\|_1$) entre la position de x et la destination b .

test: $c_{\text{hat}}(\text{board}) = 3$ car on a 3 cell mal placées

BFS est en $O(|V|+|E|)$ mais où $|E|$ est le nb de edge i.e. au max $|V|^2$

Mais dans le TP on doit dire que le BFS est exponentiel

(enfin a une base exponentielle après reste dépend de l'implémentation)
car n est le disorder lvl, i.e. nb de fois qu'on a "shuffle" avec
`gen_disorder(board, n)`.

Donc ici n va être la taille max de la solution (i.e. nb de move) i.e. la **profondeur** max. nécessaire pour **atteindre la solution** ce qui est pas du tout le nombre de edge total.

On doit aller à une profondeur n , \Rightarrow on parcourt n depth/breadth/largeur/niveau chaque largeur à entre 1 et 3 edges, on a donc que avec une recherche "aveugle" i.e. à *coût uniforme (BFS)* on parcours **au pire 3^n nodes**.

```

def isMisplaced(board: list[list[int]], coord: tuple[int, int]) → bool:
    """Returns True if the element at coord is misplaced, False otherwise. i.e. 15 should be at (3, 3)"""
    DIM = len(board)
    return board[coord[0]][coord[1]] ≠ DIM * coord[0] + coord[1] + 1

def swap(board: list[list[int]], tx: tuple[int, int], move: M, misplaced: set[tuple[int, int]] = None) → tuple[int, int]:
    """Swap index and index coord in coord in the board and update set of misplaced tiles (if not none).
    Parameters
    -----
    @ `board` - Game board
    @ `tx` - index of white square (16)
    @ `move` - move applied to the white square
    @ (optional) `misplaced` - set of misplaced tiles
    Returns: New position of the white square
    """
    nc = move + tx # new coord of white square
    board[tx[0]][tx[1]], board[nc[0]][nc[1]] = board[nc[0]][nc[1]], board[tx[0]][tx[1]] # swap
    if misplaced is None: return nc
    # if the new coord is misplaced, add it to the misplaced set
    if isMisplaced(board, nc): misplaced.add(nc)
    else: misplaced.discard(nc) # if not in it ⇒ does nothing
    # if the swap corrected the position of a tile, remove it from the misplaced set
    if isMisplaced(board, tx): misplaced.add(tx) # if already in it ⇒ does nothing
    else: misplaced.discard(tx)
    return nc

def c_hat(board: list[list[int]]) → int:
    """Cost function for a given board is the cost function the optimum, i.e.  $\hat{c}(\text{board}) = \hat{c}(\text{optimum}) = \hat{c}(\text{root})$ 
    (by relation 3.3 in the lecture notes).
    and for root,  $\hat{c}(\text{root}) = h(\text{root}) + g(\text{root}) = 0 + g(\text{root}) = g(\text{root})$  = number of misplaced tiles (16 excluded).
    Hence why we only compute  $g(\text{root})$  here.
    (in this version, everything has to be recomputed from scratch,
    for more info on "attempt" at better complexity see ``update_misplaced_compute_cost()`` below.)
    The cost of a node of the game is the number of squares that are not at their place (16 excluded)."""
    misplaced = 0
    for i in range(len(board)):
        for j in range(len(board[i])):
            if isMisplaced(board, (i, j)):
                misplaced += 1
    return max(0, misplaced - 1) # -1 because we do not count the empty square

def init_misplaced(board: list[list[int]], white_square_value) → tuple[tuple[int, int], set[tuple[int, int]]]:
    """Initialize the set of misplaced tiles and return index of white square (16).
    Parameters: `board` - Game board
    Returns: Tuple: (index of white square (16), Set of misplaced tiles)"""
    misplaced = set()
    tx0 = (-1, -1) # index of white square (16)
    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == white_square_value: tx0 = (i, j)
            if isMisplaced(board, (i, j)):
                misplaced.add((i, j))
    return tx0, misplaced

```

TAQUIN

```

def update_misplaced_compute_cost(node: Node, misplaced: set[tuple[int, int]], board: list[list[int]], moves: list[M]) → tuple[int, int, set[tuple[int, int]]]:
    """ Instead of storing copies of the board (which would be memory consuming since board is a 2d matrix),
    this function recomputes the different changes each move would have on the board, and updates the set of misplaced tiles accordingly.
    Then the cost of a node ``x`` is just its depth ('`h(x)=x.depth``') ``+`` the length of the set of misplaced tiles ('`g(x)=len(x.misplaced)`').
    the total should be in  $O(2 \times h(x))$  where  $h(x)$  is the current number of moves. (swap should be  $O(1)$  since add() and contains checks are  $O(1)$  for sets)
    Parameters
    -----
    @ `node` - Node to compute the cost of
    @ `board` - Game board
    @ `moves` - list of moves to get from the root of the game tree to this node (i.e. each "parent" edge of node)
    Returns
    -----
    Triple ``(`c(node), g(node), update_misplaced_set)`` where ``c(node) = h(node) + g(node)`` and ``update_misplaced_set`` is the updated set of misplaced tiles
    """
    _misplaced = misplaced.copy() # set of misplaced tiles

    tx = apply_moves(board, node.tx0, moves, _misplaced) # node.tx0 = index of the empty square in the initial board

    #* Reverting the swaps of Board, since it is modified to computed the updated set of misplaced tiles.
    #* making the changes and reverting them  $O(2 \times n)$  where  $n = h(x)$  is the number of Moves (which is at most  $g(\text{root})$ ) since if a solution is found the it is optimal.
    #* is still faster than making a copy of the board at each iteration.  $O(m^2)$ 
    #* (Where  $m$  is the dimension of the board, here  $m$  is a constant but we could have the exact same problem for  $m$  unknown and the same implementation would suffice)

    unapply_moves(board, tx, moves)

    gx = len(_misplaced) # g(x) = number of misplaced tiles
    hx = node.depth
    return gx+hx, gx, _misplaced

def children_moves(node: Node, DIM: int) → set[M]:
    """ Return a set of all possible moves for the children of the given node.
    @ `node` - Node to compute the cost of
    @ `DIM` - Dimension of the board
    Returns: set of all possible moves from a given node."""
    moves = M_ALL.copy()
    if node.moves != []: moves.remove(node.moves[-1].inv()) # remove inverse of last move to avoid cycles in the game "tree"
    # now remove moves that would lead to an out of bounds error
    # if row of current white square (16) is on a side (i.e. if can't go UP or DOWN)
    if node.tx[0] == 0: moves.discard(M.UP)
    if node.tx[0] == DIM-1: moves.discard(M.DOWN)
    # idem, if column of current white square (16) is on a side (i.e. if can't go LEFT or RIGHT)
    if node.tx[1] == 0: moves.discard(M.LEFT)
    if node.tx[1] == DIM-1: moves.discard(M.RIGHT)
    return moves

def P(enode: Node):
    """ Return true if and only if enode is a goal node. """
    return enode.gx == 0 # A solution is found when there is no misplaced tiles i.e. the set of misplaced tiles is empty

def addToLiveNodes(enode: Node, liveNodes: list[Node]):
    """ Adds (in place) given ``enode`` to ``liveNodes`` list, while maintaining the order. (sorted by cost  $\hat{c}$ )
    Parameters:
    `enode` - Node to add to liveNodes list
    `liveNodes` - List of live nodes (i.e. nodes that have not been expanded yet)"""
    liveNodes.append(enode)
    for i in range(len(liveNodes)-1, 0, -1): # loop backward
        if liveNodes[i].cost > liveNodes[i-1].cost: # if the cost of the node at index i is smaller than the cost of the node at index i-1, swap them
            liveNodes[i], liveNodes[i-1] = liveNodes[i-1], liveNodes[i]
        else: break # because the other elements are sorted

```

```

def nextNode(liveNodes: list[Node]) → Node:
    """ Return the next node to expand.
    Since we maintain the list sorted by cost, the next node is just ``liveNodes.pop()``.
    @ Parameters: `liveNodes` - List of live nodes (i.e. nodes that have not been expanded yet)
    @ Returns: ``liveNodes.pop()`` i.e. Node with that minimize the cost in given liveNodes list.
    """
    return liveNodes.pop()

def solve_taquin(board: list[list[int]], extract_path_from_goalNode: bool = True, white_square:int = 16) → list[str] | Node:
    """Function that solves the 15-puzzle using branch and bound,
    for the cost function ``c(x) = h(x) + g(x)`` where ``h(x) = depth of x`` and ``g(x) = number of misplaced tiles``.
    NB: technically the algorithm could work for more than 16 tiles (i.e. for a board of size n x n where n ≥ 4)
    Parameters
    -----
    @ `board` - Matrix representing the initial state of the game
    @ `convert_sol` - If True, return the list of moves as string (i.e. 'up', 'down'...) to get from the initial state to the goal state. If False, return the goal node.
    @ `white_square` - (only useful if board is a nxn matrix with n>4) Value of the white square in the initial state of the game. Default is 16.
    NB: does not modify given board
    """
    DIM = len(board) # board should be a square matrix
    liveNodes: list[Node] = []
    enode = Node(None, None, board) # special constructor for root node
    if white_square ≠ 16 and DIM > 4: enode._init_root_(board, white_square) # if white_square is not 16 then update the value used in ``init_misplaced()``
    while not P(enode):
        available_moves: set[M] = children_moves(enode, DIM)
        for move in available_moves:
            child = Node(move, enode, board)
            addToLiveNodes(child, liveNodes)
        enode = nextNode(liveNodes)
    return convert_solution(enode) if extract_path_from_goalNode else enode

```

```

        if parent is None: return # root node
        self.ak: tuple[int, int] = move + parent.ak
        self.b: tuple[int, int] = parent.b
        self.depth : int = parent.depth + 1
        self.path: list[tuple[int, int]] = parent.path + [self.ak]
        self.cost: int = Node.c_hat(self)
        self.move = move
    
```

SHORTEST PATH

```

@classmethod
def init_root(cls, a: tuple[int, int], b: tuple[int, int]):
    """ Constructor of root nodes, ``a`` is the starting point, `b` is the destination. """
    newnode = Node(None, None)
    newnode.depth = 0
    newnode.ak = a
    newnode.b = b
    newnode.cost = Node.d(a, b)
    newnode.path = [a]
    newnode.move = None
    return newnode

@classmethod
def d(cls, x1, x2) → int:
    """Returns the distance d(this, x) = ||a_k - x||_1 (i.e. norm ``L1``) between the current location and x"""
    if isinstance(x1, Node) and isinstance(x2, Node): return dist(x1.ak, x2.ak)
    if isinstance(x1, Node): return dist(x1.ak, x2)
    if isinstance(x2, Node): return dist(x1, x2.ak)
    return dist(x1, x2)

@classmethod
def c_hat(cls, node) → int:
    """Returns the cost of a node (i.e. ``c(node) = h(node) + g(node)``)"""
    return node.depth + Node.d(node, node.b)

def dist(p1: tuple[int, int], p2: tuple[int, int]):
    """Convenience function to compute the distance (L1 norm) between two points (i.e. ``||p1-p2||_1``)
    @ `p1` - first pair of coordinates
    @ `p2` - second pair of coordinates"""
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def P(enode: Node):
    """ Return true if and only if enode is a goal node. ``enode`` has arrived
    at its destination ``b`` if and only if its distance to ``b`` is null (in the matical sense)
    i.e. if ``||enode.ak - b||_1 = 0`` i.e. ``enode.ak = b``"""
    return Node.d(enode, enode.b) = 0 # A solution is found when there is no misplaced tiles i.e. the set of misplaced tiles is empty

def solve_shortest_path(domain:list[list[int]], a:tuple[int, int], b:tuple[int, int]) → list[tuple[int, int]]:
    """Finds the shortest path from point a to point b according to the 2-dimensional domain.

```

```

def solve_shortest_path(domain:list[list[int]], a:tuple[int, int], b:tuple[int, int]) → list[tuple[int, int]]:
    """Finds the shortest path from point a to point b according to the 2-dimensional domain.
    The path is returned as a list of steps from a to b, where each step is a tuple with 2 integers."""
    n, m = len(domain), len(domain[0])
    liveNodes: list[Node] = []
    enode: Node = Node.init_root(a, b)
    while not P(enode):
        available_moves: set[M] = children_moves(domain, enode, n, m)
        for move in available_moves:
            child = Node(move, enode)
            addToLiveNodes(child, liveNodes)
        enode = nextENode(liveNodes)
    return enode.path

```

Un E -node y qui succède à un E -node x est alors choisi parmi la liste des nœuds vivants selon différents critères possibles. Il est donc possible que le nouveau E -node y ne soit pas un fils de x , mais un fils d'un E -node plus ancien.

Parmi les façons de choisir le prochain E -node dans la liste des nœuds vivants, on peut citer les stratégies :

- First In First Out (FIFO) : cela veut dire que le prochain E -node est le plus ancien des nœuds vivants, ou encore que la liste des nœuds vivants est une file (*queue* en anglais).
- Last In First Out (LIFO) : le prochain E -node est le nœuds placé le plus récemment dans la liste des nœuds vivants. Cette liste est donc une structure de donnée de pile (*stack* en anglais).
- Least Cost (LC) : le prochain E -node est le nœud de la liste des nœuds vivants qui minimise une fonction de coût donnée.

Pour illustrer les deux premières stratégies, considérons l'arbre de recherche de la figure 3.1. Ci-dessous nous indiquons, entre crochets, le contenu de la liste des nœuds vivant et l'évolution au cours des itérations avec une flèche surmontée du E -node choisi.

$$\begin{array}{c}
 [0] \xrightarrow{0} [1, 2, 3] \xrightarrow{1} [2, 3, 4, 5] \xrightarrow{2} [3, 4, 5, 6, 7, 8] \\
 \xrightarrow{3} [4, 5, 6, 7, 8, 9] \xrightarrow{4} [5, 6, 7, 8, 9] \xrightarrow{5} [6, 7, 8, 9, 10, 11] \\
 \xrightarrow{6} [7, 8, 9, 10, 11] \xrightarrow{7} [8, 9, 10, 11, 12] \xrightarrow{8} [9, 10, 11, 12] \\
 \xrightarrow{9} [10, 11, 12] \xrightarrow{10} [11, 12] \xrightarrow{11} [12] \xrightarrow{12} []
 \end{array} \tag{3.1}$$

3.2 La recherche à coût minimum

La stratégie *Least Cost* nécessite d'associer une fonction de coût $c(x)$ à chaque nœud x de la structure à explorer, afin de guider la recherche vers un nœud réponse. Idéalement, $c(x)$ devrait être la profondeur minimum qui correspond au nœud réponse x^* le plus proche dans la descendance de x , et vérifiant une propriété P souhaitée. Pour illustrer ce propos, reprenons l'exemple

propriété $P(y)$ donnée et qui minimise une fonction objectif associée, $F(y)$. Dans ce cas la fonction $c(x)$ qui guide la recherche *Least Cost* devrait idéalement être la valeur minimum de F dans la descendance de x . Avec un tel choix, on aurait la garantie d'aller vers le minimum recherché par un chemin direct. On notera que cette définition de c implique que

$$c(x) \leq c(y) \tag{3.3}$$

par une fonction que l'on notera ici $\hat{c}(x)$. Les propriétés que $\hat{c}(x)$ doit vérifier pour assurer que le *Branch & Bound* trouve l'optimum recherché sont

- (1) $\hat{c}(x) \leq c(x)$ pour tout x
- (2) $\hat{c}(x) = c(x)$ pour les x qui sont une réponse (ceux pour qui la propriété $P(x)$ est vraie).

L'efficacité de la recherche, c'est-à-dire le nombre de noeuds visités avant de trouver la réponse optimale va dépendre de la qualité de l'approximation \hat{c} . Moins \hat{c} reflétera la vraie valeur de c , plus le *Branch & Bound* risque d'explorer de mauvaises branches. Mais le point important est que si les propriétés ci-dessus sont satisfaites, la recherche aboutira à coup sûr à l'optimum.

Théorème 2 *Soit une recherche Branche & Bound à coût minimum (least cost) et soit une fonction $\hat{c}(x)$ qui vérifie que $\hat{c}(x) \leq c(x)$ pour tous les nœuds x , et $\hat{c}(x^*) = c(x^*)$ pour les nœuds réponse x^* .*

On notera aussi qu'il est fréquent d'écrire $\hat{c}(x)$ de la façon suivante

$$\hat{c}(x) = h(x) + g(x) \quad (3.4)$$

où $h(x)$ est le coût déjà accumulé pour atteindre le noeud x , et $g(x)$ est une estimation optimiste (borne inférieure) du coût restant pour atteindre l'optimum recherché, x^* , dans la descendance de x . On peut bien sûr choisir $g(x) = 0$ pour tout x . On parle de recherche à *coût uniforme*. Le *Branch & Bound* trouvera l'optimum recherché, mais l'efficacité de l'exploration sera sans doute faible.

On suppose ici qu'une fonction de coût `minCost(x)` est définie pour tous nœuds x du graphe de recherche. Elle retourne un nombre réel, correspondant à la fonction $\hat{c}(x)$ définie dans la section 3.2.

Il faut bien sûr aussi spécifier la façon de représenter les nœuds x et en particulier les E -nodes successifs. Le plus général est de définir une classe `Node` qui contient un attribut `configuration` codant le noeud proprement dit (par exemple la valeur du noeud, la configuration d'un jeu, la position d'un robot dans un graphe, etc). Ensuite, il est utile d'avoir aussi un attribut `cost` associé à chaque noeud, donnant le coût de ce noeud. Puis, il peut être nécessaire de sauvegarder le chemin qui a été emprunté de la racine x_0 de l'exploration jusqu'à un noeud courant x . Cela est par exemple réalisable par un attribut `path` qui est une liste des nœuds visités entre x_0 et x , ou encore de mouvements choisis pour le parcours effectué. Cet attribut est par exemple essentiel dans les cas où on cherche les mouvements nécessaires à atteindre un but x^* à partir d'une configuration initiale x_0 donnée. Un tel exemple est mentionné dans la section 3.4.

triée dans l'ordre des coûts (décroissant) => nextNode est juste liveNodelist.pop()

Il faut aussi définir une fonction `listOfChildren(node)` qui génère la liste des noeuds enfants d'un noeud `node`. On stockera la liste des nœuds vivants dans la liste `liveNodelist` et on supposera qu'il existe une fonction

```
addToLiveNodelist(node, liveNodelist)
```

qui ajoute le noeud `node` à la liste des nœuds vivants en maintenant cette dernière triée par ordre décroissant des coûts. Un exemple est donné dans le code de l'algorithme 16, où un tri par insertion est utilisé.

Algorithm 16 Implémentation de `addToLiveNodeList(child, liveNodeList)` afin de maintenir la liste triée dans l'ordre décroissant des coûts. Les nouveaux nœuds sont insérés dans la liste jusqu'à ce que le nœud précédent ait un coût plus élevé.

```
1 def addToLiveNodeList( child , liveNodeList ):  
2     liveNodeList . append( child )  
3     for i in range( len( liveNodeList )-1 , 0 , -1 ): # loop backward  
4         c1=liveNodeList [ i ]. cost  
5         c2=liveNodeList [ i -1 ]. cost  
6         if c1>c2:  
7             ( liveNodeList [ i ] , liveNodeList [ i -1 ] )=(  
8                 liveNodeList [ i -1 ] , liveNodeList [ i ] )  
9         else: break # because the other elements are sorted
```

Maintenir la liste des nœuds vivants triée permet de facilement construire la fonction `nextENode(liveNodeList)` qui retourne le prochain *E*-node. Celui-ci est le nœud de la liste `liveNodeList` de coût minimum, c'est-à-dire le dernier de cette liste triée, comme indiqué sur le listing de l'algorithme 17.

Algorithm 17 Listing de la fonction `nextENode(liveNodeList)`

```
1 def nextENode( liveNodeList ):  
2     node=liveNodeList . pop()  
3     return node
```

Finalement, il faut définir une fonction `P(node)` qui retourne `True` si `node` est une réponse. L'algorithme 18 propose une implémentation possible du *Branch & Bound* à coût minimum. Cet algorithme s'arrête dès qu'une solution est trouvée. Par le théorème 2, c'est une solution optimale.

Algorithm 18 Implémentation du *Branch & Bound*

```
1 liveNodeList=[]  
2 E_node=root # root is the initial point  
3 while not P(E_node):  
4     l=listOfChildren( E_node ) # liste intermédiaire  
5     for child in l: addToLiveNodeList( child , liveNodeList )  
6     E_node=nextENode( liveNodeList )  
7             # liveNodeList.pop()  
8     print "Optimal solution:" ,  
9     printNode( E_node )
```

^ maintient ordre décroissant

On sort de la boucle => `P(E_node) == True`
=> `E_node` est un nœud réponse (voir thm 2 pour preuve)

Exemple simple

Pour donner un exemple simple des parties manquantes de l'implémentation ci-dessus, on peut imaginer le problème de trouver la chaîne de 3 bits qui contient le minimum de bit à 1. La réponse est évidemment la chaîne 000, mais on va montrer comment le *Branch & Bound* fonctionne dans un tel cas. A chaque étage de l'arbre de recherche on ajoute un bit de plus à la chaîne. Il y a chaque fois deux branches possibles car on peut ajouter le bit 0 ou le bit 1. Les noeuds sont représentés par l'attribut `path` qui est la succession des bits choisis. Il n'est donc pas nécessaire d'ajouter un attribut «configuration» pour ce problème car il serait identique à `path`.

L'implémentation de la classe `Node` est donnée à l'algorithme 19. Un noeud

Algorithm 19 Implémentation de la classe `Node` pour un cas où on utilise que deux attributs, `cost` et `path`

```
1 class Node:
2     def __init__(self, Cost, Path):
3         self.cost=Cost
4         self.path=Path[:] Path[:] === copy (pas view, mais pas deep non plus, littéralement equiv. à Path.copy())
```

réponse est un noeud dont les 3 bits sont spécifiés, ainsi qu'indiqué dans le listing de l'algorithme 20.

Algorithm 20 Listing de la fonction `P(node)` pour un problème où la profondeur de l'arbre détermine si un noeud est réponse ou non

```
1 depth=3
2 def P(node, depth):
3     if len(node.path)==depth: return True # return node.x >= 2**depth
4     return False
ou plus simplement :
1 def P(node, depth):
2     return len(node.path)==depth
```

La fonction de coût $c(x)$ est le nombre de bits à 1 du noeud de l'arbre qui en contient le moins dans la descendance de x . Pour le noeud racine, c vaut 0, en référence au noeud réponse 000. La fonction $\hat{c}(x)$ est simplement le nombre de bits déjà fixés à 1 dans le noeud x . Dans cet exemple on a donc que $c(x) = \hat{c}(x)$. Cette fonction de coût est utilisée dans l'algorithme 21 qui implémente la fonction `listOfChildren(node)`. Cette fonction construit une

liste avec les enfants du *E-node*. Il y en a deux pour ce problème : celui dont le prochain bit est 0 et dont le coût est égal au coût du *E-node*, et celui dont le prochain bit est 1 et dont le coût est égal au coût du *E-node* additionné de 1.

Algorithm 21 Listing de la fonction `listOfChildren(node)` pour le problème de trouver la chaîne de bit avec le moins de 1 possibles.

```

1  def listOfChildren ( node ) :
2      l = []
3      Path = node . path [ : ]
4      Path . append ( 0 )
5      child = Node ( node . cost , Path )
6      l . append ( child )
7      Path = node . path [ : ]
8      Path . append ( 1 )
9      child = Node ( node . cost + 1 , Path )
10     l . append ( child )
11     return l

```

```

def listOfChildren(node):
    l = []
    Path = node.path[:] # chemin du parent
    Path.append(0) # branche de gauche
    x = 2*node.x + 0 # fils de gauche
    cost = node.cost + W[x] - Wmin
    child = Node(x, Cost, Path)
    l.append(child)
    :
    :
    :
    2'eme enfant à droite
    Path.append(1)
    x = 2*node.x + 1
    :
    :
    :
    return l

```

L'exécution du *Branch & Bound* sur ce problème donne le résultat ci-dessous, avec la succession des *E-nodes* choisis et le contenu de la liste de noeuds vivants. On voit que la recherche va au plus vite vers la solution optimale, le noeud 000 de coût minimum.

```

E-node: (path= cost=0)
liveNodeList: [(path=1 cost=1) (path=0 cost=0)]
E-node: (path=0 cost=0)
liveNodeList: [(path=1 cost=1) (path=01 cost=1) (path=00 cost=0)]
E-node: (path=00 cost=0)
liveNodeList: [(path=1 cost=1) (path=01 cost=1) (path=001 cost=1) (path=000 cost=0)]
E-node: (path=000 cost=0)
Optimal solution: (path=000 cost=0)

```

TAQUIN EXEMPLE:

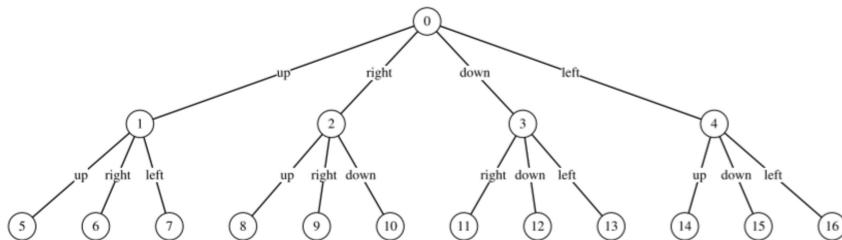


FIGURE 3.6 – Deux niveaux successifs des mouvements possibles du jeu du taquin.

cases à leur place, car un mouvement ne bouge qu'une case². Par exemple dans la configuration de gauche de la figure 3.5, seules les cases 1 et 11 sont à leur place et on aura donc

$$g(x) = 13$$

La figure 3.8 illustre l'arbre des mouvements possibles à partir de la même configuration initiale que celle utilisée dans la figure 3.7. La configuration qui porte le numéro 22 est la réponse. On va voir maintenant qu'elle est trouvée par le *Branch & Bound* de façon optimale, à savoir par la succession des mouvements down→right→down. Pour s'en convaincre on va regarder en détail l'évolution de la liste des noeuds vivants et du choix des *E-node* successifs. En désignant les noeuds par le couple

$$(x, \hat{c}(x)) = (x, h(x) + g(x))$$

cases à leur place, car un mouvement ne bouge qu'une case². Par exemple dans la configuration de gauche de la figure 3.5, seules les cases 1 et 11 sont à leur place et on aura donc

$$g(x) = 13$$

La figure 3.8 illustre l'arbre des mouvements possibles à partir de la même configuration initiale que celle utilisée dans la figure 3.7. La configuration qui porte le numéro 22 est la réponse. On va voir maintenant qu'elle est trouvée par le *Branch & Bound* de façon optimale, à savoir par la succession des mouvements down→right→down. Pour s'en convaincre on va regarder en détail l'évolution de la liste des noeuds vivants et du choix des *E-node* successifs. En désignant les noeuds par le couple

$$(x, \hat{c}(x)) = (x, h(x) + g(x))$$

où x est le numéro indiqué au-dessus des configurations présentées sur la figure 3.8, on obtient, en choisissant comme *E-node* le noeud de coût minimum de la liste

$$\begin{array}{l}
 \text{h + g} \\
 \text{ID config, } \leq [(0, 0 + 3)] \xrightarrow{0} [(1, 1 + 4), (2, 1 + 4), (3, 1 + 2), (4, 1 + 4)] \\
 \text{g (nb de cases} \\
 \text{mal placés)} \xrightarrow{3} [(1, 1 + 4), (2, 1 + 4), (4, 1 + 4), (9, 2 + 1), (10, 2 + 3), (11, 2 + 3)] \\
 \xrightarrow{9} [(1, 1 + 4), (2, 1 + 4), (4, 1 + 4), \\
 \quad (10, 2 + 3), (11, 2 + 3), (21, 3 + 2), (22, 3 + 0)] \\
 \xrightarrow{22} \text{solution}
 \end{array} \tag{3.8}$$

Comme prévu par le théorème 2, la solution accessible avec le moins de mouvements est trouvée. On constate aussi que ce chemin optimal est découvert en trois étapes seulement. Pour la condition initiale considérée ici, la fonction \hat{c} choisie a guidé la recherche de façon directe sur l'optimum recherché.