```
algorithm backtrack():
  if (solution == True)
    return True                          <-- Solution found
  for each possible moves
    if(this move is valid)
      select this move and place
      ok = call backtrack()              <-- Keep exploring
      if ok:
          return solution found
      unplace that selected move
  return False                           <-- Don't explore anymore! no solution in this path
```

```
def branch_and_bound():
    activeset = [root_node]
    best_solution = None

    while(activset not empty):
        choose the most promising node k from activeset
        remove the node k from activeset
        generate the children of node k and estimate its lower and upper bound
        for each child i of node k:
            if (upper bound of child i is worse than bestval):
                kill child i
            elif(child i is a complete solution):
                bestval = solution
            else:
                add child i to activeset
```

```
K = [2,3,4,6,5]

list(subset_sum(K, 11))
```

```
[[6, 5], [2, 4, 5], [2, 3, 6]]
```

```
def subset_sum(array, num):
    if num < 0:
        return
    if len(array) == 0:
        if num == 0:
            yield []
        return
    for solution in twentyone(array[1:], num):
        yield solution
    for solution in twentyone(array[1:], num - array[0]):
        yield [array[0]] + solution
```

```
from queue import PriorityQueue

def inf_bound(matrix):
   # Mínim de cada columna
    if len(matrix)==0:
        return 0
    return sum(matrix.min(axis=0))

def sup_bound(matrix):
    # Assignació qualsevol. En aquest cas sumem la diagonal
    # que consisteix a assignar
    # la tasca 'i' a l'empresa 'i' on i=0,1,2...
    return sum(matrix.diagonal()) if len(matrix)!=0 else 0
```

```
def sum_K(lst, K):
    sum_K_backtracking(lst, K, 0, 0, [])
```

```
def sum_K_backtracking(lst, K, tmp_sum, idx, sub_list):
    # Comprovacions inicials
    if K < 0:
        print("no solution")
        return False

    if len(lst) == 0:
        if K == 0:
            print("[]")
            return True
        print("no numbers to create subset")
        return False

    # Condició de parada, si la suma acumulada és igual al valor de la
    if (tmp_sum == K):
        print(sub_list)
        return True

    # for each possible moves
    for n in range(idx, len(lst)):

        # Si satisfà la condició
        if(lst[n] + tmp_sum <= K):
            # select this move and place
            sub_list.append(lst[n])
            tmp_sum += lst[n]
            idx += 1

            # keep exploring (crida recursiva)
            ok = sum_K_backtracking(lst, K, tmp_sum, n+1, sub_list)

            # re-inicialize variables
            if ok:
                sub_list.remove(lst[n])
                tmp_sum -= lst[n]

            else:
                # unpalace not valid move
                sub_list.remove(lst[n])
                tmp_sum -= lst[n]
```

```
def tasks(matrix):
    # Cotes inicials
    sup = sup_bound(matrix)
    inf = inf_bound(matrix)

    # Cua de prioritat. Guardarem quatre elements:
    # 1. Prioritat
    # 2. Parelles ja assignades (tasca, empresa)
    # 3. Tasca que hem d'assignar a continuació (row)
    # 4. Empreses ja assignades (col)
    pq = PriorityQueue()
    pq.put((inf, [], 0, set([])))

    # Iterarem mentre la cua de prioritat no sigui buida
    while not pq.empty():

        # Extraiem un element
        elem_cota, elem_list, elem_row, elem_cols = pq.get()

        # Podem assignar la tasca 'elem_row' a qualsevol empresa 'col' que no haguem assignat encara
        for col in range(len(matrix)):
            if col not in elem_cols:

                # Copiem els originals ja que els modificarem
                new_elem_list, new_elem_cols = elem_list.copy(), elem_cols.copy()

                # Afegim els nous elements a la llista de visitats i afegim la parella
                new_elem_cols.add(col)                # Afegim l'empresa seleccionada
                new_elem_list.append((elem_row, col)) # Afegim una nova parella
                new_elem_row = elem_row + 1           # Indiquem que haurem de continuar amb una nova tasca

                # OPCIONAL: Ep! Si hem assignat la penúltima, l'última ja ens ve determinada
                if len(new_elem_list) == len(matrix)-1:

                    # La tasca (erow) serà l'última (longitud de la matriu - 1)
                    # L'empresa (ecol) serà la que no estigui dins el conjunt d'empreses assignades
                    erow, ecol = len(matrix)-1, list(set(range(len(matrix))) - new_elem_cols)[0]
                    new_elem_cols.add(ecol)
                    new_elem_list.append((erow, ecol))
                    new_elem_row += 1

                # Com fem per calcular la cota?
                # Eliminem de la matriu les files i columnes que ja haguem usat (ens quedem amb les NO assignades)
                matrix_slice = np.delete(matrix, list(range(0,new_elem_row)), 0) # Files, fins la fila que ens pertoca assignar
                matrix_slice = np.delete(matrix_slice, list(new_elem_cols), 1) # Columnes, eliminem les columnes ja assignades

                # Calculem la cota amb la suma de parelles assignades + el mínim de la matriu restant
                new_elem_cota = sum(matrix[i,j] for i,j in new_elem_list) + inf_bound(matrix_slice)

                # Mirem si és solució. Això passarà si tenim les mateixes parelles dins la llista
                # que la mida de la matriu
                if len(new_elem_list) == len(matrix):

                    # En cas que haguem trobat una cota millor, imprimim i actualitzem la cota
                    # (l'últim print que es faci serà la millor opció)
                    if new_elem_cota < sup:
                        print("Millor solució trobada, cost:", new_elem_cota)
                        print("Actualitzem la cota superior de", sup, "a", new_elem_cota)

                        sup = new_elem_cota

                        print("Assignacions: ")
                        for i, j in new_elem_list:
                            print('Tasca',i,'-> Empresa',j)
                        print("-"*60)

                # Altrament, només l'afegim si potencialment ens millora la cota
                elif new_elem_cota < sup:
                    pq.put((new_elem_cota, new_elem_list, new_elem_row, new_elem_cols))
```

```
costs = np.array([[11,12,18,40],
                  [14,15,13,22],
                  [11,17,19,23],
                  [17,14,20,28]])

print("Matriu de costs:")
print(costs)
print()

tasks(costs)
```

```
import numpy as np

def solve_puzzle(board):
    # Millor solució trobada. Inicialment té cota infinit
    best_bound = np.inf
    best_board = board

    # Guardem en una cua de prioritat els taulells.
    # Guardarem les variables:
    # 1. Distància mínima (cota inferior) entre el tauler actual i el tauler solució
    # 2. Número de passes que duem en aquest tauler, g(X)
    # 3. El tauler
    pq = PriorityQueue()
    pq.put((board.manhattan_distance(), 0, board))

    # Com que els estats poden repetir-se al llarg de l'exploració, guardarem en un 'set' tots els
    # estats visitats. Així evidem tornar a visitar estats.
    existent_states = set([board.get_state_id()])
    expanded = 0

    while not pq.empty():

        # Obtenim un nou element de la cua
        curr_bound, curr_steps, curr_board = pq.get()
        expanded += 1

        # Mirem tots els moviments valids que podem fer des d'aquest tauler
        for a_move in curr_board.allowed_moves():
            new_board = curr_board.move(a_move)
            new_steps = curr_steps + 1
            new_bound = new_steps + new_board.manhattan_distance() # g(X) + h(X)

            # Si és un estat solució i ens millora la cota, actualitzem.
            if new_board.state():
                if new_bound < best_bound:
                    best_bound = new_bound
                    best_board = new_board

            # En cas de que no sigui solució però ens millori la cota.
            elif (new_bound < best_bound) and (new_board.get_state_id() not in existent_states):
                existent_states.add(new_board.get_state_id())
                pq.put((new_bound,new_steps,new_board))

    return best_bound, best_board, expanded
```

```python
def solve_deck(N):
    solution = solve_deck_backtracking(N, [0]*(2*N), set([]))

    if not solution:
        return f"N={N:<2}: No s'ha trobat solució"
    return f"N={N:<2}: {solution}"
def solve_deck_backtracking(N, solution, placed_nums):
    # Cas base. Considerem que la solucio no te més zeros, hem acabat!
    if 0 not in solution:
        return True
    # També podria ser:
    # if(len(placed_nums) == N):
    #     return True

    # Provem de posar el següent número
    for n in range(N,0,-1):

        # Filtrem els números que ja hem col·locat prèviament
        if n not in placed_nums:

            # Busquem el primer índex on poguem posar-hi un valor a
            # la llista de solution
            idx1 = solution.index(0)

            # Busquem el segon índex (ha d'haver-hi n cartes enmig)
            idx2 = idx1 + n + 1

            # idx1 ja sabem que té un zero, però idx2 podria no tenir un 0
            # o bé sortir-se del taulell (mida 2*N)
            # Cal comprovar que el segon index existeixi i estigui buit!
            if idx2 < 2*N and solution[idx2]==0:

                # En el cas que estigui tot correcte, afegim a la llista
                # de números utilitzats i modifiquem la solució.
                placed_nums.add(n)
                solution[idx1], solution[idx2] = n, n

                # Cridem recursivament mentre tot estigui correcte
                ok = solve_deck_backtracking(N, solution, placed_nums)

                if ok:
                    # Tot perfecte, acabem
                    return solution

                # No ha funcionat, desfem el moviment
                placed_nums.remove(n)
                solution[idx1], solution[idx2] = 0, 0

    # No hi ha cap més número que poguem posar, retornem False
    return False
```

```python
def solve_queens_backtracking(N, board, col):
    # Les columnes van indexades de 0,...,N-1.
    # Si arribem a una més, hem acabat
    if col==N:
        return True

    # Estem posant la reina de la columna 'col'
    # Provem totes les files.
    for row in range(N):

        # Si satisfà les restriccions
        if check_position_previous_columns(board, row, col):

            # Posem la reina
            board[row][col] = 1

            # Cridem recursivament
            ok = solve_queens_backtracking(N, board, col+1)

            # Si se satisfà la condició, hem trobat la solució!
            if ok:
                return board

            # Si no, traiem la reina del lloc on es troba
            board[row][col] = 0
    return False
def solve_queens(N):
    board = [[0]*N for _ in range(N)]
    solution = solve_queens_backtracking(N, board,0)

    if not solution:
        return f'N={N}: No té solució'
    return f'N={N}:\n{format_board(solution)}'
def format_board(board):
    _str = "+"
    for i in board[0]:
        _str += "---+"
    _str += "\n"
    for i in board:
        _str += "|"
        for j in i:
            _str += "   |" if j == 0 else " Q |"
        _str +="\n+"
        for j in i:
            _str += "---+"
        _str +="\n"
    return _str
```

```python
def knapSack(mW,w,v,n):
    # counter how many time recursive function is called.
    global c
    c += 1

    if(mW == 0 or n == 0):
        return [0,[]]

    if(w[n-1] > mW):
        return knapSack(mW,w,v,n-1)

    set1 = knapSack(mW-w[n-1],w,v,n-1)
    set2 = knapSack(mW,w,v,n-1)

    if(set1[0]+v[n-1] > set2[0]):
        set1[1].append(n-1)
        set1[0] += v[n-1]
        return set1
    else:
        return set2

val = [160, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print("Knapsack Max & list:",knapSack(W, wt, val, n))
```

```python
def check_position_previous_columns(board, row, col):
    # Comprovem les columnes anteriors de la fila 'row'
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Comprovem la diagonal inferior
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Comprovem la diagonal superior
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True
```

```python
max_weight = 23
item_values = [16, 15, 4, 3, 2]
item_weights =[14, 13, 7, 2, 1]
num_items = len(item_weights)

def knapsack(max_weight, item_values, item_weights, num_items,
             current_weight, current_value, index, items,
             best_weight, best_value, best_items):
    # Hem trobat una solució.
    if index == num_items:
        # Si la solució trobada és millor que l'anterior, actualizem.
        if current_value > best_value:
            best_weight, best_value, best_items = current_weight, current_value, items
            print("SOLUTION: ",best_weight, best_value, best_items, items, index)
        return best_weight, best_value, best_items

    # Possibles moviments - > Posem l'ítem o no el posem
    for (add_weight, add_value, add_item) in  [(0, 0, -1), (item_weights[index], item_values[index],
        if current_weight + add_weight <= max_weight: # si el moviment es vàlid
            current_weight += add_weight
            current_value += add_value
            index += 1
            items.append(add_item)

            best_weight, best_value, best_items = knapsack(max_weight, item_values, item_weights, num_items,
                                                           current_weight, current_value, index, items,
                                                           best_weight, best_value, best_items)

            index -= 1
            current_value -= add_value
            current_weight -= add_weight
            items.pop()
    return best_weight, best_value, best_items

knapsack(max_weight, item_values, item_weights, num_items, 0, 0, 0, [], 0, 0, [])
```

```
SOLUTION:  1 2 [-1, -1, -1, -1, 4] [-1, -1, -1, -1, 4] 5
```

```python
def solve(board,i,j):
    # Solution found
    if(i==9):
        printBoard(board)
        return True
    # La casella ja conté un número
    # Anem a la següent
    if board[i][j] != 0:
        if j == 8:
            solve(board,i+1,0)
        else:
            solve(board,i,j+1)
    else:
        for val in range(1,10):
            if isPossible(board,i,j,val):
                # Select move
                board[i][j] = val

                if j == 8:
                    solve(board,i+1,0)
                else:
                    solve(board,i,j+1)

                # Bad choice, unplace
                board[i][j] = 0
        return False
    # We found a solution, print it
```

```python
def map_painting(city_colors, cities, adjMatrix, index, colors):
    # if a solution has been found
    if(index== len(cities)):
        printSolution(city_colors)
        return True

    for c in colors:
        if(valid_movement(adjMatrix, city_colors, c, index)):
            city_colors[index] = c
            if(map_painting(city_colors, cities, adjMatrix, index+1, colors)):
                return True
            city_colors[index] = 0
    return False


colors = ['R','G','B']
cities = ['WA','NT','SA','Q','NSW','V','T']
city_colors = [0 for i in range(len(cities))]

adjMatrix = [[ 0, 1, 1, 0, 0, 0, 0],
             [ 1, 0, 1, 1, 0, 0, 0],
             [ 1, 1, 0, 1, 1, 1, 0],
             [ 0, 1, 1, 0, 1, 0, 0],
             [ 0, 0, 1, 1, 0, 1, 0],
             [ 0, 0, 1, 0, 1, 0, 1],
             [ 0, 0, 0, 0, 0, 1, 0]]

map_painting(city_colors, cities, adjMatrix, 0, colors)
def printSolution(results):
    print("Solution Exists:" " Following are the assigned colors ")
    for i in range(len(results)):
        print(results[i],end=" ")


def check_if_solution_valid(adjMatrix, solution):
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            if (adjMatrix[i][j] and solution[j] == solution[i]):
                return False
    return True
```

```python
def printBoard(board):
    print("-"*37)
    for i, row in enumerate(board):
        print(("|" + " {}   {}   {} |"*3).format(*[x if x != 0 else "0" for x in row]))
        if i == 8:
            print("-"*37)
        elif i % 3 == 2:
            print("|" + "---+"*8 + "---|")
        else:
            print("|" + "   +"*8 + "   |")

def isPossible(board,row,col,val):
    # Val already in row
    for x in range(9):
        if board[row][x] == val:
            return False

    # Val already in col
    for x in range(9):
        if board[x][col] == val:
            return False

    # Val already in submatrix
    startRow = row - row%3
    startCol = col - col%3

    for i in range(3):
        for j in range(3):
            if board[i+startRow][j+startCol] == val:
                return False
    return True
```

```python
board = [[0, 0, 0, 8, 0, 0, 4, 0, 3],
         [2, 0, 0, 0, 0, 4, 8, 9, 0],
         [0, 9, 0, 0, 0, 0, 0, 0, 2],
         [0, 0, 0, 0, 2, 9, 0, 1, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 7, 0, 6, 5, 0, 0, 0, 0],
         [9, 0, 0, 0, 0, 0, 0, 8, 0],
         [0, 6, 2, 7, 0, 0, 0, 0, 1],
         [4, 0, 3, 0, 0, 6, 0, 0, 0]]

print("TAULER INICIAL")
printBoard(board)

print('\n')
solve(board,0,0)
```