O(1)   def euclidean_distance(town1:tuple, town2:tuple):

O(1)   return ((town1[0] - town2[0])**2 + (town1[1] - town2[1])**2)**0.5

O(N+N*(N+N)) = O(N^2)

                def glouton(size:int, towns:dict):

O(N)   to_visit = [i for i in range(1, size)]

O(1)   result = [0]

O(N)   while len(to_visit) > 0:

O(1)   min_found, index = float('inf'), float('inf')

O(N)   for i in to_visit:

O(1)   distance = euclidean_distance(towns[result[-1]], towns[i])

O(1)   if distance < min_found:

O(1)   min_found, index = distance, i

O(1)   result.append(index)

O(N)   to_visit.remove(index)

O(1)   return result

O(n log(n)) car selon théorème maitre k=1, b=1 et l=1. Plus formellement T(n) = 1*T(n-1)+ 2*(n^1)

            def powerset(seq):

O(1)   if len(seq) <= 1:

O(1)   yield seq

O(1)   yield []

                else:

O(N-1)   for item in powerset(seq[1:]):

O(1)   yield [seq[0]]+item

O(1)   yield item

O(2N + N! + N*log(N) + N + (N-1)!*N*(N-1) + N + N + N + N) = O(N!*N^2)

            def progdyn(size:int, towns:dict):

```
O(N)            town_indexes = list(towns.keys())
O(N)            town_indexes.remove(0)
O(N!+N*log(N)) = (next line)
O(N!)           cols = [set(x) for x in powerset(town_indexes)]
O(N*log(N))     cols.sort(key=lambda x: len(x))
O(1)            cols.pop()
O(1)            dyn_table = dict()

O(N)            for k in town_indexes:
O(1)                dyn_table[(k, frozenset(cols[0]))] = euclidean_distance(towns[0], towns[k]), 0

O((N-1)!)       for subset in cols[1:]:
O(N)                for k in town_indexes:
O(1)                    if k in subset:
O(1)                        dyn_table[(k, frozenset(subset))] = None
                    else:
O(1)                        min_found, min_found_index = float('inf'), float('inf')
O(N-1)                      for j in subset:
O(1)                            distance = euclidean_distance(towns[k], towns[j]) + dyn_table[(j, frozenset(subset - {j}))][0]
O(1)                            if distance < min_found:
O(1)                                min_found, min_found_index = distance, j
O(1)                        dyn_table[(k, frozenset(subset))] = min_found, min_found_index

O(1)            min_path, min_path_index = float('inf'), float('inf')
O(N)            town_indexes_set = set(town_indexes)
O(N)            for k in town_indexes_set:
O(1)                distance = euclidean_distance(towns[0], towns[k]) + dyn_table[(k, frozenset(town_indexes_set - {k}))][0]
O(1)                if distance < min_path:
O(1)                    min_path, min_path_index = distance, k
```

O(N) car on passe dans toutes les valeurs de town_indexes_set avec une taille égale à N

```
                def get_path(k:int, town_indexes_set:set):
O(1)                if len(town_indexes_set) == 1:
O(1)                    return [k]
O(1)                return [k] + get_path(dyn_table[(k, frozenset(town_indexes_set - {k}))][1], town_indexes_set - {k})


O(N)            result = get_path(min_path_index, town_indexes_set)
O(1)            result.append(0)
O(N)            result.reverse()
O(1)            return result


            class Graph():
O(1)            def __init__(self, vertices, towns:dict):
O(1)                self.V = vertices
O(1)                self.nodes = {}
O(1)                self.towns = towns
```

O(N+N*M) = O(N*M) où M est le nombre d'enfant par noeuds

```
                def printMST(self, parent):
O(N)                for i in range(1, self.V):
O(1)                    if parent[i] not in self.nodes:
O(1)                        self.nodes[parent[i]] = []
O(1)                        self.nodes[parent[i]].append({"child":i, "weight":euclidean_distance(self.towns[i],
                                                                    self.towns[parent[i]])})


O(1)                stack = [0]
O(1)                result = []
O(N)                while len(stack) > 0:
O(1)                    current = stack.pop()
O(1)                    result.append(current)
O(1)                    if current in self.nodes:
O(M)                        for child in self.nodes[current]:
```

```
O(1)                    stack.append(child["child"])
O(1)              return result

O(N)            def minKey(self, key, mstSet):
O(1)              min = sys.maxsize

O(N)              for v in range(self.V):
O(1)                if key[v] < min and mstSet[v] == False:
O(1)                  min = key[v]
O(1)                  min_index = v

O(1)              return min_index


O(N*(N+N) + N*M) = O(N^2), car N*M < N^2 car M < N
             def primMST(self):
O(1)             key = [sys.maxsize] * self.V
O(1)             parent = [None] * self.V
O(1)             key[0] = 0
O(1)             mstSet = [False] * self.V
O(1)             parent[0] = -1
O(N)             for cout in range(self.V):
O(N)               u = self.minKey(key, mstSet)
O(1)               mstSet[u] = True
O(N)               for v in range(self.V):
O(1)                 uv_distance = euclidean_distance(self.towns[u], self.towns[v])
O(1)                 if uv_distance > 0 and mstSet[v] == False \ and key[v] > uv_distance:
O(1)                   key[v] = uv_distance
O(1)                   parent[v] = u

O(N*M)            return self.printMST(parent)
```

O(N^2)   def approx(size:int, towns:dict):

O(1)   g = Graph(size, towns)

O(N^2)   return g.primMST()