Le problème étudié dans ce TP est le suivant. Sur les bords d'une rivière il y n missionnaires et n cannibales. Ils disposent d'une barque pouvant contenir p personnes ($p < 2n$). La question est de savoir si toutes les personnes peuvent traverser sachant qu'il ne doit jamais y avoir plus de cannibales que de missionnaires sur une des rives ou sur la barque (quand la barque arrive à une rive on considère que tout le monde descend de la barque).

On va repérer les états (qui vont correspondre aux sommets du graphe) par la situation sur la rive initiale. Un état est donc un triplet de type `int*int*int` où le premier entier est le nombre de missionnaires, le deuxième le nombre de cannibales et le troisième désigne la position de la barque : s'il vaut `0` la barque est au niveau de rive de départ et s'il vaut `1` elle est sur l'autre rive. Le sommet initial est donc le sommet `(n,n,0)`.

On stockera donc les états dans un tableau de longueur $2(n+1)^2$ de telle sorte que le sommet (x, y, z) soit dans la case d'indice $i = (n+1)x + y + (n+1)^2 z$.

1) Écrire une fonction `sommetToTriplet` de type `int -> int -> int*int*int` définie de telle sorte que `sommetToTriplet n i` renvoie le triplet `x,y,z` correspondant au sommet d'indice `i` du tableau.

2) On veut écrire une fonction `transports` de type `int -> int -> int -> int -> (int*int) list` tel que `transports n p x y` renvoie la liste de toutes les compositions possibles (le couple donné par le nombre de missionnaires et le nombre de cannibales) de la barque sachant que l'on suppose que la barque est au bord d'une rive où il y a `x` missionnaires et `y` cannibales. On fera attention à
   - Il doit rester plus de missionnaires que de cannibales (au sens large) sur la rive dont part la barque (s'il reste des missionnaires)
   - Il ne doit pas avoir plus de p personnes sur la barque
   - Il ne doit pas y avoir strictement plus de cannibales que de missionnaires dans la barque (s'il y a des missionnaires)
   - Il ne doit pas y avoir strictement plus de cannibales que de missionnaires quand les personnes de la barque descendent sur l'autre rive (s'ils y a des missionnaires).

   Écrire la fonction `transports`. On pourra vérifier que `transport 3 2 3 3` renvoie (à l'ordre près) :

   ```
   - : (int * int) list = [0, 2; 1, 1; 0, 1]
   ```

3) On veut construire le graphe modélisant le problème.
   - Les sommets qui sont les situations décrites ci-dessus sont représentés par les entiers $i = (n+1)x + y + (n+1)^2 z$.
   - Les arêtes vont d'un sommet (x, y, z) vers $(x - (-1)^z a, y - (-1)^z b, 1 - z)$ pour chaque voyage de barque remplie avec a missionnaires et b cannibales.

   Écrire une fonction `graphe : int -> int -> int list array` tel que `graphe n p` construit le graphe défini par listes d'adjacences. On pourra vérifier que `graphe 3 2` renvoie

   ```
   - : int list array =
    [|[]; [16]; [17; 16]; [18; 17]; [16]; [17; 16]; [18; 21; 17]; [19; 18; 21];
      [16]; [21; 17]; [18; 21]; [19; 26]; []; [21; 28]; [26; 29; 28];
      [30; 26; 29]; [1; 5; 2]; [5; 2; 3]; [10; 3]; []; [12; 5]; [13; 10];
      [10; 14]; [15]; [12; 13; 10]; [13; 10; 14]; [14; 15]; [15]; [13; 14];
      [14; 15]; [15]; []|]
   ```

4) a) A l'aide d'un parcours de graphe écrire une fonction `possible` de type `int -> int -> bool` telle que `possible n p` renvoie un booléen selon qu'il est possible de faire traverser les missionnaires et les cannibales. On pourra tester `possible 3 2` et `possible 4 2`.

   b) Écrire une fonction `solution n p` de type `int -> int -> (int*int) list` telle que `solution n p` renvoie une solution du problème. Elle renverra une liste de couples sachant que chaque couple désigne le nombre de missionnaires et le nombres de cannibales de chaque trajet. Si ce n'est pas possible la fonction lèvera une exception. A-t-on la solution qui nécessite le moins de transports ? On pourra vérifier que `solution 3 2` renvoie une solution en 11 voyages.

5) Un autre algorithme - l'algorithme A* On veut mettre en place un autre algorithme afin déterminer plus rapidement une éventuelle solution. Le principe de l'algorithme A* est le suivant.

On dispose d'un état initial (ici tous les missionnaires, les cannibales et le bateau sur la rive de départ) que l'on note i et un état terminal (ici tous les missionnaires, les cannibales et le bateau sur la rive d'arrivée) que l'on note t.

On a alors un ensemble d'états F (pour frontière) qui sont les états que l'on a vus mais pas encore visités et un ensemble V d'états visités.

On a un tableau de prédécesseur `prec` qui associera à chaque sommet un prédécesseur.

On suppose de plus disposer de :

- une fonction g qui associe à tout état x le « cout » $g(x)$ pour aller de i à x. Dans notre cas ce sera le nombre de trajets en bateaux pour aller de i à x. Notons que l'on ne calcule cette fonction que lorsque l'on peut atteindre x.
- une fonction h qui est une heuristique qui **évalue** le « cout » pour aller de x à t [1]. Nous prendrons $h(x) = \dfrac{n_x + n_c}{2}$ où n_x et n_c désignent le nombre de missionnaires et de cannibales sur la rive de départ.

On pose alors f la fonction qui associe à un sommet x la valeur $g(x) + h(x)$ qui est le cout estimé pour aller de i à t en passant par x.

L'algorithme est alors le suivant

- On initialise V à vide ; on initialise F au sommet initial ; on a donc $g(i) = 0$ et $f(i) = h(i)$.
- Tant que F n'est pas vide :
  - On enlève de F l'élément x tel que $f(x)$ est minimal.
  - On insère x dans V (si $x = t$ la solution est trouvée c'est fini !)
  - Pour tout voisin y de x qui n'est pas déjà dans $F \cup V$:
    - On insère y dans F
    - $g(y) \leftarrow g(x) + 1$
    - $f(y) \leftarrow g(y) + h(y)$
    - `prec(y) <- x`

Écrire une fonction `solutionAstar` de type `int -> int -> ((int*int*int)-> int) -> (int*int) list` telle que `solutionAstar n p h` où `h` est l'heuristique renvoie une solution (en supposant qu'elle existe). En fonction du temps on réfléchira ou pas à une structure de données adaptée pour gérer l'ensemble F.

Remarque : Dans la pratique, il est souvent plus efficace de ne pas commencer en déterminant l'intégralité du graphe et de ne calculer les voisins que pour les sommets que l'on atteint lors de l'exploration. En particulier dans les cas où l'espace des états (les sommets du graphe) devient très grand. L'intérêt alors de l'algorithme A* est de ne pas parcourir (et donc calculer) les parties du graphe éloignées de la solution cherchée.

1. Pour ne pas avoir de problèmes, l'heuristique ne doit pas surestimer la distance ; c'est-à-dire que l'heuristique $h(x)$ doit être inférieure au cout pour aller de x à t