Exercice 1. Le problème de la somme des sous-ensembles

Question 1 – L'ensemble S est l'ensemble des sous-listes de `li` dont la somme des éléments vaut n. Le but est de construire l'un des éléments de S. À chaque étape, on s'intéresse à l'un des entiers e de `li`. Il y a alors deux choix possibles :

- ⋆ On ajoute e à la liste en construction.
- ⋆ On n'ajoute pas e à la liste en construction.

Question 2 – La fonction `backtrack` renvoie `None` s'il n'y a pas de solution, sinon elle renvoie `Some li` avec `li` la sous-liste solution.

```
let subset_sum (li0: int list) (s0: int) =
  let rec backtrack li s = match (li,s) with
    | _ , 0 -> Some []
    | [], _ -> None
    | e :: q, _ ->
       match backtrack q (s-e) with
       | Some li1 -> Some (e::li1)
       | None -> backtrack q s
  in
  match backtrack li0 s0 with
  | Some li -> li
  | None -> failwith "Pas de solution";;
```

Question 3 – L'arbre des appels récursifs pour la fonction `backtrack` est un sous-arbre d'un arbre binaire complet de hauteur n. Ainsi :

La complexité temporelle est en $\mathcal{O}(2^n)$.

La complexité spatiale est en $\Theta(n) + \Theta(n) = \Theta(n)$ (hauteur de l'arbre + mémoire nécessaire pour stocker les listes).

Exercice 2. Le problème des n dames

Question 1 – On place la dame de la ligne 0, puis celle de la ligne 1, et ainsi de suite jusqu'à la dame de la ligne $n-1$.

```
(* Indique si deux dames en positions (i1,j1) et (i2,j2) sont en conflit.
   Suppose i1 <> i2 *)
let conflit i1 j1 i2 j2: bool =
  j1 = j2 || i1+j1 = i2+j2 || i1-j1 = i2-j2;;
```

```
(* Indique s'il est possible de placer une dame en position (i1,j1).
   Pour i < i1, pos.(i) est la colonne de la dame numéro i. Pour i >= i1, les
   valeurs pos.(i) sont arbitraires. *)
let rec pos_valide i1 j1 (pos: int array) =
  let i = ref 0 in
  while !i < i1 && not (conflit i1 j1 !i pos.(!i)) do
    incr i;
  done;
  !i = i1;;
```

```
(* Suppose que les dames < i sont déjà placées.
   Renvoie un booléen indiquant s'il est possible de placer la dame i sur une
   colonne >= j, puis de placer les dames > i.
   Les positions des dames sont stockées dans pos *)
let rec backtrack_dames i j pos =
  let n = Array.length pos in
  if i = n then true
  else if j = n then false
  else if not (pos_valide i j pos) then backtrack_dames i (j+1) pos
  else begin pos.(i) <- j;
             if backtrack_dames (i+1) 0 pos then true
             else backtrack_dames i (j+1) pos;
       end;;
```

```
let sol_dames (n: int) =
  let pos = Array.make n 0 in
  if backtrack_dames 0 0 pos then pos else failwith "Pas de solution";;
```

Question 2 – On adapte les fonctions de la question précédente.

```
let rec backtrack_dames_nb_sols i j pos =
  let n = Array.length pos in
  if i = n then 1
  else if j = n then 0
  else if not (pos_valide i j pos) then backtrack_dames_nb_sols i (j+1) pos
  else begin pos.(i) <- j;
             (* Effets de bord: l'ordre des appels rec est important ! *)
             let n1 = backtrack_dames_nb_sols (i+1) 0 pos in
             let n2 = backtrack_dames_nb_sols i (j+1) pos in
             n1 + n2;
    end;;
```

```
let nb_sols_dames (n: int) =
  let pos = Array.make n 0 in
  backtrack_dames_nb_sols 0 0 pos;;
```

Question 3 – On s'intéresse à l'arbre des appels récursifs de la fonction `backtrack`. On remarque que lorsqu'on l'appelle avec un couple `(i,j)`, il y a au plus deux appels récursifs avec des couples `(i',j')` tels que `(i,j) < (i',j')` (pour l'ordre lexicographique). De plus, il n'y a pas d'appel récursif lorsque `i = n` ou `j = n`.

Dans l'arbre des appels récursifs, la racine est étiquetée par `(i, j) = (0, 0)`. Lorsqu'on emprunte un chemin vers une feuille, tous les noeuds internes sont étiquetés par des couples `(i, j)` $\in [\![0, n-1]\!]^2$. Ainsi,

l'arbre des appels récursifs est binaire et de hauteur majorée par n^2. Donc son nombre de noeuds est en $\mathcal{O}(2^{n^2})$.

→ La complexité spatiale est de la forme $\mathcal{O}(n) + \mathcal{O}(n^2) = \boxed{\mathcal{O}(n^2)}$ (mémoire nécessaire pour stocker le tableau + hauteur de l'arbre des appels récursifs).

→ Pour la complexité temporelle, on remarque que sans compter les appels récursifs, chaque appel à la fonction `backtrack` est en $\mathcal{O}(n)$ (appel à la fonction `est_valide`). Ainsi le temps d'exécution est en $\boxed{\mathcal{O}(n2^{n^2})}$.

Exercice 3. Cavalier d'Euler

Question 1 – On note `(i0,j0)` les coordonnées de la case initiale. On utilise une matrice `mat` de taille $n \times n$ telle que `mat.(i).(j)` vaut `-1` si le cavalier n'est pas passé sur la case d'indice `(i,j)` et sinon vaut l'étape à laquelle est passé le cavalier.

→ La fonction `cases_suiv` renvoie la liste de toutes les cases non visitées accessibles depuis la case `(i,j)`.

→ La fonction `backtrack` prend en entrée une liste de cases et indique s'il est possible de terminer le tour d'Euler en partant de l'une de ces cases (attention : elle modifie également le tableau en entrée).

```
let cases_suiv (i: int) (j: int) (tour: int array array) =
  let n = Array.length tour in
  let li = [(i+1, j+2); (i+1, j-2); (i-1, j+2); (i-1, j-2);
            (i+2, j+1); (i+2, j-1); (i-2, j+1); (i-2, j-1)] in
  let test (k,l) =
    k >= 0 && k < n && l >= 0 && l < n && tour.(k).(l) = -1
  in
  List.filter test li;;
```

```
(* TC = tour de cavalier *)
let trouver_TC (i0: int) (j0: int) (n: int) =
  let tour = Array.make_matrix n n (-1) in
  let rec backtrack li k =
    if k = n*n then true else
      match li with
      | [] -> false
      | (i,j) :: q -> tour.(i).(j) <- k;
                      if backtrack (cases_suiv i j tour) (k+1) then true
                      else (tour.(i).(j) <- -1; backtrack q k)
  in
  if backtrack [(i0,j0)] 0 then tour else failwith "Pas de tour possible";;
```

Question 2 – On adapte la fonction de la question précédente.

```
let nb_TC (i0: int) (j0: int) (n: int) =
  let tour = Array.make_matrix n n (-1) in
  let cpt = ref 0 in
  let rec backtrack li k =
    if k = n*n then incr cpt else
      match li with
      | [] -> ()
      | (i,j) :: q ->
         tour.(i).(j) <- k;
         backtrack (cases_suiv i j tour) (k+1);
         tour.(i).(j) <- -1;
         backtrack q k;
  in
  backtrack [(i0,j0)] 0;
  !cpt;;
```

Question 3 – Si on trace l'arbre des appels récursifs de la fonction `backtrack`, on a un arbre de profondeur $n^2 - 1$ où chaque noeud a au plus 8 fils.

La complexité temporelle est en $\mathcal{O}(8^{n^2})$.

La complexité spatiale est en $\Theta(n^2) + \Theta(n^2) = \Theta(n^2)$ (hauteur de l'arbre des appels récursifs + mémoire nécessaire pour stocker la matrice).

Remarque. Pour terminer, on présente une deuxième solution beaucoup plus rapide, mais plus difficile à mettre en oeuvre.

Question 1. Après avoir visité une case c, pour chaque voisin v de c, on note n_v le nombre de voisins de v qui n'ont pas encore été visitées (c est considéré comme déjà visité et n'est pas comptabilisé dans n_v). On remarque que :

- Si l'un des n_v est égal à 0 et qu'on ne se trouve pas à la dernière étape, alors le tour ne pourra pas être complété.
- Si l'un des n_v vaut 1, alors on peut forcer le tour à visiter v juste après c. Il n'est donc pas nécessaire de lancer les appels récursifs sur les autres voisins de c.

  En effet, s'il n'existe pas de tour qui visite v juste après c, alors il n'existe pas de tour du tout. Pour le montrer, supposons qu'il existe un tour T, mais qu'il ne visite pas v juste après c. Le dernier sommet de T est nécessairement v (puisqu'il n'a qu'un voisin non visité). Ainsi, en inversant l'ordre des cases empruntées, on obtient un tour dans lequel on a visité v juste après c.

Un moyen simple de prendre en compte les deux points ci-dessus est d'utiliser l'heuristique suivante : à chaque étape, on visite les voisins de c par ordre croissant de n_v.

Question 2. On reprend les notations ci-dessus. Si l'un des n_v vaut 1, alors pour tout $v' \neq v$ les deux entiers suivants sont égaux :

- Le nombre de tours qui visitent v juste après c et dont le dernier sommet est v'.
- Le nombre de tours qui visitent v' juste après c. Notez que le dernier sommet de ces tours est nécessairement v, puisqu'il n'a qu'un voisin non visité.

Ainsi, lorsque l'un des n_v vaut 1, on force le tour à visiter v juste après c et on calcule une matrice `res` telle que `res.(i).(j)` est le nombre de tours dont la dernière case est `(i,j)`. Alors, pour chaque voisin `(i,j)` de c qui n'est pas v, le nombre de chemins qui visitent `(i,j)` juste après c est égal à `res.(i).(j)`.

Cette astuce permet de lancer un seul appel récursif : celui sur v (les appels récursifs sur les voisins $v' \neq v$ de c sont inutiles puisque leurs résultats se déduisent de celui sur v).

Exercice 4. Le problème des n dames (avec des exceptions)

Question 1 – On place la dame de la ligne 0, puis celle de la ligne 1, et ainsi de suite jusqu'à la dame de la ligne $n-1$. La fonction `pos_valide` indique s'il est possible de placer une dame en position `(i1,j1)` (la liste donnée en entrée contient les positions des dames déjà placées).

```
let rec pos_valide_exn i1 j1 = function
  | [] -> true
  | (i2,j2) :: _ when j1 = j2 -> false
  | (i2,j2) :: q when i1+j1 = i2+j2 -> false
  | (i2,j2) :: q when i1-j1 = i2-j2 -> false
  | _ :: q -> pos_valide_exn i1 j1 q;;
```

```
let sol_dames_exn n =
  let rec backtrack li i =
    if i = n then raise (Fini li);
    for j = 0 to n-1 do
      if pos_valide_exn i j li then backtrack ((i,j)::li) (i+1)
    done;
  in
  try (backtrack [] 0; failwith "Pas de solution") with
  | Fini li -> li;;
```

Exercice 5. Carrés magiques

Question 1 – On appelle « groupe de cases » un ensemble de cases dont la somme doit être égale à $\frac{n(n^2+1)}{2}$ (c'est à dire une ligne, une colonne ou l'une des deux diagonales). Les groupes de cases vont être représentés par des listes contenant les coordonnées `(i,j)` des cases.

À chaque étape de l'algorithme, on sélectionne la première case vide et on essaye d'y mettre successivement tous les nombres qui n'apparaissent pas encore dans la grille tout en respectant les conditions du carré magique.

```
(* g.cases.(i).(j) = 0 si la case est vide
   Pour k > 0: g.nombres.(k) vaut true si k apparait dans la grille
   Pour k = 0: g.nombres.(k) est arbitraire *)
type grille = {cases: int array array;
               nombres: bool array};;
```

```
(* k = 0 autorisé dans cette fonction *)
let remplir_case (g: grille) (i: int) (j: int) (k: int): unit =
  g.nombres.(g.cases.(i).(j)) <- false;
  g.cases.(i).(j) <- k;
  g.nombres.(k) <- true;;
```

```
(* gc = groupes de cases *)
let somme_gc (g: grille) (gc: (int*int) list) =
  List.fold_left (fun a (i,j) -> a + g.cases.(i).(j)) 0 gc;;
```

```
let nb_zeros (g: grille) (gc: (int*int) list): int =
  List.fold_left (fun a (i,j) -> a + if g.cases.(i).(j) = 0 then 1 else 0) 0 gc;;
```

```
(* Renvoie deux listes contenant tous les groupes de cases auxquels appartient
   (i0,j0). La première liste contient les groupes de cases dans lesquels il ne
   reste qu'une case vide. La deuxième liste contient ceux dans lesquels il
   reste aux moins deux cases vides. *)
let liste_gc (g: grille) (i0: int) (j0: int) =
  let n = Array.length g.cases in
  let li_gc = ref [] in
  li_gc := List.init n (fun j -> i0,j) :: !li_gc;
  li_gc := List.init n (fun i -> i,j0) :: !li_gc;
  if i0 = j0 then li_gc := List.init n (fun i -> i,i) :: !li_gc;
  if j0 = n-1-i0 then li_gc := List.init n (fun i -> i,n-1-i) :: !li_gc;
  List.filter (fun gc -> nb_zeros g gc = 1) !li_gc,
  List.filter (fun gc -> nb_zeros g gc <> 1) !li_gc;;
```

```
(* cp = coup possible *)
let est_cp g i j k =
  let m = Array.length g.nombres - 1 in
  let li_gc1, li_gc2 = liste_gc g i j in
  not g.nombres.(k) &&
    List.for_all (fun gc -> k + somme_gc g gc = m ) li_gc1 &&
    List.for_all (fun gc -> k + somme_gc g gc <= m) li_gc2 ;;
```

```
(* ij = i*n+j *)
let rec backtrack (g: grille) (ij: int) (k: int) =
  let n = Array.length g.cases in
  let i = ij/n and j = ij mod n in
  if ij = n*n then true
  else if k = n*n+1 then false
  else if g.cases.(i).(j) <> 0 then backtrack g (ij+1) 1
  else if not (est_cp g i j k) then backtrack g ij (k+1)
  else begin remplir_case g i j k;
             if backtrack g (ij+1) 1 then true
             else begin remplir_case g i j 0;
                        backtrack g ij (k+1);
                  end;
       end;;
```

```
let carre_magique (cases0: int array array) =
  let n = Array.length cases0 in
  let m = (n*(n*n+1))/2 in
  let g = {cases = Array.make_matrix n n 0;
           nombres = Array.make (m+1) false} in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      let k = cases0.(i).(j) in
      if k <> 0 && not (est_cp g i j k) then
        failwith "La grille initale est incohérente";
      remplir_case g i j k;
    done;
  done;
  if backtrack g 0 1 then g.cases else failwith "Pas de solution";;
```

Remarque. On peut obtenir un algorithme plus rapide en vérifiant s'il y a des coups forcés : si un groupe de cases ne contient qu'une seule case vide, alors on peut remplir directement cette case.