TD n°14 – Recherche exhaustive

Exercice 1 *Le mot de passe*

Le mot de passe de Jean est formé de 4 caractères qui sont soit des lettres minuscules, soit des lettres amjuscules, soit des chiffres. (Vous aurez besoin de la table ASCII, regardez sur internet)

On vous donne une fonction `bool est_correct(char* mdp)` qui indique si le mot de passe proposé est le mot de passe de Jean.

Écrire une fonction `char* bruteforce()` qui trouve le mot de passe de Jean par la force brute.

Ici l'ensemble des candidats est A^4 où A désigne l'ensemble des lettres et des chiffres. On peut utiliser la fonction suivant ou 4 boucles for. La principale difficulté est d'énumérer les lettres et les chiffres, car ils ne se suivent pas dans la table ASCII.

Pour ce faire on crée une fonction `char charassocie(int i)` qui associe aux nombres entre 0 et 61 un caractère.

```
char charassocie(int i){
   if (0<=i<=9){
     return i+48; //Le chiffre 0 est de code ASCII 48
   }
   else if (10<=i<=35){
     return i-10+65; //La lettre A est de code ASCII 65
   }
   else{
     return i-36+97; //La lettre a est de code ASCII 97
   }
}

char* bruteforce(){
   char* mdp = malloc(5*sizeof(char));
   mdp[4] = '\0';
   for (int c1=0;c1<=61;c1+=1){
     mdp[0]=charassocie(c1);
     for (int c2=0;c2<=61;c2+=1){
       mdp[1]=charassocie(c2);
       for (int c3=0;c3<=61;c4+=1){
         mdp[2]=charassocie(c3);
         for (int c4=0;c4<=61;c4+=1){
           mdp[3]=charassocie(c4);
           if(est_correct(mdp)){return mdp;}
         }}}}}
```

Exercice 2 *Un diner presque parfait*

Alice a invité $n \in \mathbb{N}$ personnes à son anniversaire, mais ces personnes ne s'entendent pas toutes ensemble.

Elle a réservé un restaurant avec une grande table ronde, et elle voudrait répartir les convives (elle y compris) de sorte à ce que deux personnes qui se détestent ne soient pas côte à côte.

Les convives seront numérotés de 0 à N (Alice est le numéro 0). On vous donne une fonction `deteste : int->int->bool` qui indique si la personne a déteste la personne b (la détestation est réciproque).

Le plan de table sera représenté par une liste de taille $N+1$. Il faut se souvenir que la tête est voisine du dernier élément puisque la table est ronde.

1. Écrire une fonction `est_plan : 'a list -> bool` qui prend en entrée un plan de table t et teste si ce plan de table ne contient pas deux voisins qui se détestent.

```
let est_plan l =
  let rec aux l p = (*p est le premier convive, pour comparer avec le dernier*)
    match l with
      |[e] -> not (deteste p e)
      |t1::t2::q -> not (deteste t1 t2) && (aux (t2::q) p)
  in match l with
    |[]|[e] -> true
    |t1::t2::q -> not (deteste t1 t2) && (aux (t2::q) t1);;
```

Pour énumérer tous les plans de table possible, on peut procéder récursivement.

2. Écrire une fonction `plan_de_table_naif : int -> 'a list` qui prend en entrée N et renvoie un plan de table qui fonctionne en suivant une méthode par force brute.
   Remarque : les plans de table possibles sont des permutations de $[|0, N-1|]$, c'est à dire des manières d'ordonner les nombres entre 0 et $N-1$.

En suivant le principe expliqué à la lettre, il s'avère qu'on a deux choix :

- Soit générer la liste de toutes les permutations.
- Soit tester chaque permutation individuellement quand on a fini de la construire, mais cela nécessite des instructions de la forme **`l@[e]`** (ajout d'un élément à droite), ce que je me refuse à faire.

```
let plan_de_table_naif n =
    let rec ajoute_en_tete e l = (*Ajoute e en tête de tous les éléments de l*)
      match l with
        |[] -> []
        |t::q -> (e::t)::(ajoute_en_tete e q) in

    let rec genere_permutations prec reste =
        (*génere la liste de toutes les permutations en choisissant un par un chaque
        convive pour être le premier élément de la permutation.
        prec est la liste des convives qu'on a déja testés.
        reste est la liste des convives qu'on va tester*)
        match prec, reste with
            |[],[] -> [[]] (*0 convives*)
            |[],[e] -> [[e]] (*1 convive*)
            |_,[] -> [] (*On a déja testé tous les convives*)
            |_,t::q ->
              (ajoute_en_tete t (genere_permutations [] (prec@q))) @ (genere_permutations (t::prec) q)
            (*On ajoute : - les permutations obtenues avec l'élément t comme premier élément (c'est
                          a dire les permutations de prec@reste sur lesquelle on rajoute t en tete
                          - les permutations obtenues avec les éléments qu'on a pas encore testés
                          comme premier élément (t devient un élément déja testé)*)
    in

    let rec itere_est_plan l = (*itere est_plan sur une liste de plans de table et compte combien marchent*)
      match l with
      |[] -> 0
      |t::q -> if est_plan t then 1+(itere_est_plan q) else itere_est_plan q
    in

    let rec range i = match i with (*Crée la liste [0;1;...; i]*)
      |0 -> [0]
      |_ -> i::(range (i-1))
    in

    let toutes_permutations = genere_permutations [] (range (n-1)) in (*la liste de toutes les permutations*
    itere_est_plan toutes_permutations;; (*On compte le nombre de permutations qui marchent dans la liste de
```

Si on veut résoudre l'exercice sans créer la liste de toutes les permutations et sans utiliser `@` pour concaténer avec un liste de taille 1, il faut suivre le principe expliqué dans le TD, mais dans l'autre sens : on choisit le dernier élément de la permutation d'abord, et on construit une permutation des éléments restants en ajoutant en tête les éléments au fur et à mesure.

```
  let plan_de_table_naif n =
    let rec test_permutations prec reste perm =
      (* Entrées : perm la permutation déja construite.
                   prec les éléments qu'on a déja testé de rajouter sur perm.
                   reste ceux qu'on doit encore tester.

      Cette fonction renvoie le nombre de permutations qui sont des plans de table valides
      parmi les permutations de la forme truc@perm, avec truc une permutation quelconque des
      éléments qui ne sont pas dans perm (c'est a dire des éléments de prec@reste)*)
      match prec, reste with
        |[],[] -> if est_plan perm then 1 else 0
        |_,[] -> 0
        |_,t::q -> test_permutations [] (prec@q) (t::perm)  + (test_permutations (t::prec) q perm)
        (*On ajoute : - le nombre de solutions obtenues avec un permutation de la forme truc,t,perm
                      - le nombre de solutions obtenues avec une permutation de la forme truc,x,perm
                        avec x dans q *)
    in
    let rec range i = match i with (*Crée la liste [0;1;...; i]*)
      |0 -> [0]
      |_ -> i::(range (i-1))
    in
    test_permutations [] (range(n-1)) [];;
```

Exercice 3 *Gestion de troupeau extraterrestre*

Sur une planète longue et fine (qu'on assimilera au segment $[|0, N|]$) vit un fermier extraterrestre. Il possède n vaches, réparties sur la planète à des coordonnées entières $v_i \in [|0, N|]$ et n piquets, plantés à des coordonnées entières $p_i \in [|0, N|]$. Les vaches sont peu actives et restent en permanence à leur position.

Le fermier veut attacher ses vaches à des piquets, de sorte à minimiser la longueur de corde nécessaire. Chaque piquet ne peut être attaché qu'à une seule vache (et réciproquement).

1. Écrire une fonction `solution_exhaustive : int array -> int array -> int array` qui prend en entrée le tableau des positions des vaches, le tableau des positions des piquets et renvoie un tableau de taille n, qui indique à la case i à quel piquet la vache i est attachée dans une solution optimale.

   Ici une solution c'est une manière d'associer à chaque piquet une vache, c'est à dire un tableau dans lequel la case i contient le numéro de la vache associée au piquet i. Avec cette manière de voir les choses, il est impossible de donner le même piquet à deux vaches, par contre il faut faire attention à ne pas mettre la même vache à deux piquets.

   On peut aussi dire que les solutions sont des permutations des vaches. Avec cette vision, générer les solutions se fait de la même manière que dans l'exercice précédent (avec des tableaux par contre).

   Pour diversifier la correction, on va dire qu'on ne sait pas trop ce qu'est une permutation et qu'on a vu ça comme des p-uplets qui n'ont pas de doublons. (à l'avenir je ne recommande pas de faire ça mais un des buts de cet exercice est de chercher sans être trop guidé)

```
let a_doublon tab = (*teste si le tableau tab a un doublon*)
  let res = ref false in
  for i = 0 to Array.length tab -1 do
    for j = i+1 to Array.length tab -1 do
      if tab.(i)=tab.(j) then
        res := true
    done;
  done;
  !res;;

let suivant tab = (*La fonction qu'on connait bien,
  change tab en le puplet suivant et indique si c'est la derniere configuration*)
  let n = Array.length tab in
  let i = ref 0 in
  while !i<n && (tab.(!i)=n-1) do
    tab.(i)<-0;
    i:=!i+1
  done;
  if !i=n then false
  else begin
    tab.(i)<-tab.(i)+1;
    true
  end;;

let distance tab v p = (*Calcule la longueur de corde associée a une solution*)
  let n = Array.length tab in
  let res = ref 0 in
  for i=0 to n-1 do
    res:=!res+abs(p.(i)-v.(tab.(i)))
  done;
  !res;;

let solution_exhaustive v p =
  let n = Array.length v in
  let min = ref (n*n) in
  let solmin = [||] in
  let tab = Array.make n 0 in

  while suivant tab do
    if not (a_doublon tab) then (*On ne regarde pas les puplets qui ont des doublons*)
      let d = distance tab p v in
      if d<!min then begin
        min:=d;
        solmin:=Array.copy tab (*Il faut copier le tableau sinon suivant va le modifier*)
      end
```

2. Réfléchir à une méthode par balayage qui permet de résoudre le problème en $O(nlog(n))$.
   Intuitivement on a envie de dire que le piquet le plus à gauche va avec la vache la plus à gauche, et ainsi de suite. Il faudrait le montrer pour en être sûr.
   On voudrait donc "balayer" la planète de gauche à droite en associant les piquets et les vaches rencontrées. Pour faire cela, il faut qu'on trie les piquets et les vaches par leur position, pour avoir un accès facile à ceux qui sont le "plus à gauche".
   Trier coûte $O(nlog(n))$ (on le suppose) et balayer coute $O(n)$.

Exercice 4 *Nombre d'occurences d'un mot dans un texte*

Proposer une méthode pour déterminer le nombre d'occurences d'un mot m dans un texte t, en C.

Il est recommandé de réfléchir à votre méthode par morceaux, en réfléchissant aux fonctions intermédiaires dont vous pourriez avoir besoin. On pourra, en bonus, proposer des améliorations à la méthode par force brute.

On note l la longueur du mot m.

La méthode naive est : pour chaque position i du texte, vérifier si les caractères entre $t[i]$ et $t[i+l]$ forment le mot m.

Beaucoup d'améliorations sont possibles, on verra des algorithmes bien plus performants en fin d'année.

Des améliorations simples (mais qui changent peu la complexité) :

Si le texte est en bon français, séparer les mots selon les espaces et regarder si un des mots est m.

Si le texte est en bon français, choisir la lettre du mot qui apparait le moins en français, regarder dans t si cette lettre apparait et si oui, est-ce qu'on peut reformer le mot m autour de cette occurence.