Correction du DS 5

Julien Reichert

La correction des questions de cours étant notamment dans le cours ou dans les corrigés de TD, elle ne sera pas donnée ici...

Problème

Première année

Question P1.1 :

Plusieurs approches sont possibles pour résoudre ce problème : on peut voir ceci comme une recherche de plus court chemin dans un graphe dont les sommets sont les sommes payées (dans un intervalle délimité de manière pertinente) et les arcs correspondent au paiement ou au rendu d'une pièce particulière, on peut aussi mettre en œuvre un algorithme de programmation dynamique qui serait globalement équivalent à un parcours en largeur du graphe évoqué, mais en pratique au vu du système monétaire la réponse est immédiate : on convertit la somme à payer en gallions, mornilles et noises, et on regarde le meilleur choix entre payer le nombre de gallions demandé ou ce nombre plus un en attendant la monnaie en mornilles (sachant qu'il faut anticiper la suite pour savoir ce qui est le mieux entre les deux choix), pour laquelle l'approche est la même, et on termine par les noises.

```
int mini(int a, int b)
{
  return a < b ? a : b;
}

int rendu (int somme)
{
  int noises = somme % 29;
  int m_g = somme / 29;
  int mornilles = m_g % 17;
  int gallions = m_g / 17;
  int t1 = gallions + mornilles + noises;
  int t2 = gallions + 1 + (17 - mornilles) + noises;
  int t3 = gallions + 1 + (17 - mornilles - 1) + (29 - noises);
  int t4 = gallions + mornilles + 1 + (29 - noises);
  return mini(mini(t1, t2), mini(t3, t4));
}
```

Question P1.2 :

La méthode directe revient à prendre successivement chacun des $4n$ éléments de la liste de départ et toutes les répartitions acceptables des éléments suivants, puis d'ajouter l'élément considéré à toute liste de taille strictement inférieure à n, démultipliant ainsi les scénarios.

La fonction qui subira le `fold_left` a été créée localement plutôt que d'utiliser `fun` ou de la créer globalement, et elle utilise une référence de liste par flemme de faire une récursion et surtout de changer le type après coup.

La taille de liste obtenue est exponentielle (voir la question suivante) donc inutile d'optimiser.

```
let repartitions l =
  let n = List.length l / 4 in (* pas de vérification *)
  let rec repartit reste = match reste with
  | [] -> [(([], [], [], []), (0, 0, 0, 0))]
  | a::q ->
    let toutes_repartitions_et_tailles = repartit q in
    let ajoute_a ((l1, l2, l3, l4), (n1, n2, n3, n4)) =
    let res = ref [] in
    if n4 < n then res := ((l1, l2, l3, a::l4), (n1, n2, n3, n4+1)) :: !res;
    if n3 < n then res := ((l1, l2, a::l3, l4), (n1, n2, n3+1, n4)) :: !res;
    if n2 < n then res := ((l1, a::l2, l3, l4), (n1, n2+1, n3, n4)) :: !res;
    if n1 < n then res := ((a::l1, l2, l3, l4), (n1+1, n2, n3, n4)) :: !res;
    !res
    in List.fold_left (fun buff repar -> buff @ ajoute_a repar) [] toutes_repartitions_et_tailles
  in List.map fst (repartit l);;
```

Question P1.2bis :

D'après le cours de combinatoire, le nombre de répartitions (qui s'obtient de la même manière que le nombre de façon de répartir un paquet de cartes en quatre mains) d'un ensemble de taille $4n$ en quatre ensembles de taille n est $\binom{4n}{n}\binom{3n}{n}\binom{2n}{n}$, qui est aussi égal à $\frac{(4n)!}{(n!)^4}$.

On peut justifier le calcul en disant qu'on fait trois récupérations successives d'un sous-ensemble de taille n parmi un ensemble qui se restreint de plus en plus au fur et à mesure (la troisième extraction donne le quatrième sous-ensemble en tant que le reste).

Deuxième année

Question P2.1 :

Pour minimiser la distance maximale, il faut apparier les occurrences de chaque caractère en respectant l'ordre d'apparition dans les chaînes (donc le premier `'L'` de la première chaîne avec le premier `'L'` de la deuxième chaîne, le deuxième avec le deuxième, etc.).

Comme c'est gratuit, on vérifie dans la foulée que la deuxième chaîne est une anagramme de la première.

```
let distmaxmin s1 s2 =
  let dico = Hashtbl.create 8 in
  for i = 0 to String.length s1 - 1 do
    match Hashtbl.find_opt dico s1.[i] with
    | None -> Hashtbl.add dico s1.[i] ([i], [])
    | Some (l1, _) -> Hashtbl.replace dico s1.[i] (i::l1, [])
  done;
  for i = 0 to String.length s2 - 1 do
    match Hashtbl.find_opt dico s2.[i] with
    | None -> failwith "Vraiment pas des anagrammes !"
    | Some (l1, l2) -> Hashtbl.replace dico s2.[i] (l1, i::l2)
  done;
  let rec mouline_cle couple = match couple with
  | [], _ | _, [] -> failwith "Pas des anagrammes !"
  | [a], [b] -> abs(b-a)
  | a::q, b::qq -> max (abs(b-a)) (mouline_cle (q, qq))
  in Hashtbl.fold (fun carac couple maximum -> max (mouline_cle couple) maximum) dico 0;;
```

Question P2.1bis :

Modifications à apporter : double boucle sur des tableaux de chaînes ; dans la boucle extérieure, baliser la boucle intérieure entre le premier et le dernier caractère hors espacements s'il y en a ; dans la liste, mettre les coordonnées au lieu du seuil indice ; la valeur absolue de la différence devient la racine de la somme des carrés des différences des deux coordonnées ; l'ordre d'apparition n'étant plus forcément pertinent, il faut tester toutes les permutations de la deuxième liste et garder le plus petit maximum.

Question P2.2 :

```
void recopie(int* depuis_ou, int* vers_ou)
{
  for (int i = 0 ; i <= depuis_ou[0] ; i += 1) vers_ou[i] = depuis_ou[i];
}

int* minsup(int s, int* t, int n)
{
  int opti = 0;
  for (int i = 0 ; i < n ; i += 1) opti += t[i];
  if (opti <= s) exit(1);
  int* rep = malloc((n+1)*sizeof(int)); rep[0] = 0;
  bool* possibles = malloc((s+1)*sizeof(bool));
  for (int i = 1 ; i <= s ; i += 1) possibles[i] = false;
  possibles[0] = true;
  int** comment = malloc((s+1)*sizeof(int*));
  for (int i = 0 ; i <= s ; i += 1)
  {
    comment[i] = malloc((n+1)*sizeof(int)); comment[i][0] = 0;
  }
  for (int indice = 0 ; indice < n ; indice += 1)
  {
    for (int valeur = s ; valeur >= 0 ; valeur -= 1)
    {
      if (!possibles[valeur]) continue;
      int vbis = valeur + t[indice];
      if (vbis <= s && !possibles[vbis])
      {
        possibles[vbis] = true;
        recopie(comment[valeur], comment[vbis]);
        comment[vbis][0] += 1;
        comment[vbis][comment[vbis][0]] = indice;
      }
      else if (vbis > s && vbis < opti)
      {
        opti = vbis;
        recopie(comment[valeur], rep);
        rep[0] += 1;
        rep[rep[0]] = indice;
      }
    }
  }
  for (int i = 0 ; i <= s ; i += 1) free(comment[i]);
  free(comment);
  free(possibles);
  return rep;
}
```

Il s'agit d'un problème de programmation dynamique, qu'on peut en fait aussi résoudre par exploration exhaustive (le choix serait certes plus pertinent avec une belle récursion en OCaml). Pour le programme proposé ici, chaque valeur inférieure ou égale au seuil sera assortie d'une information booléenne indiquant si la valeur peut être obtenue et d'un tableau précisant comment, en collectant les indices dans le tableau en argument où prendre les valeurs (cela donne un découpage particulier, le premier à avoir été trouvé, on ne minimise pas le nombre d'éléments à utiliser). Par suite, une variable mémorise la plus petite valeur strictement supérieure au seuil qui puisse être réalisée et un tableau, qui sera renvoyé, détermine comment faire selon la même convention.

Attention : L'ordre des boucles et le sens de la boucle intérieure ont été définis ainsi pour éviter qu'une valeur ne se répète.

Troisième année

Question P3.1 :

Dans le TD sur la programmation dynamique, le problème d'ordonnancement de tâches pondérées était traité. Il s'avère que ce problème admet une variante où le nombre d'agents pour effectuer les tâches est supérieur à un (mais fixe). Le problème qui nous concerne se résout par exemple en lançant la résolution de cette variante pour un nombre d'agents qui augmente d'un à chaque essai jusqu'à ce que le seuil soit atteint, le nombre d'agents étant alors la réponse.

Pour résoudre le problème d'ordonnancement de tâches pondérées avec un nombre fixé d'agents $n \geq 1$, on procède comme dans la version à un agent : on trie les tâches par fin croissante et on regarde dans l'ordre s'il vaut mieux les incorporer ou non. La différence est que si on les incorpore, il faut décider à quel agent les attribuer, et le choix sera de prendre celui qui doit renoncer à un ensemble de tâches de la plus petite valeur pour pouvoir faire la nouvelle tâche. Inutile de regarder alors parmi les égalités celui qui est « libre depuis le plus longtemps » de par le tri préalable.

```
type cours = { debut : int; fin : int; satisfaction : int };;

let inferieur cours1 cours2 =
  cours1.fin < cours2.fin || cours1.fin = cours2.fin && cours1.satisfaction > cours2.satisfaction ||
  cours1.fin = cours2.fin && cours1.satisfaction = cours2.satisfaction && cours1.debut < cours2.debut;;

let split tab debut fin =
  let indice = ref (debut+1) and indice_fin = ref fin and pivot = tab.(debut) in
    while !indice <= !indice_fin do
      if inferieur tab.(!indice) pivot then incr indice
      else
        begin
          let buff = tab.(!indice) in
          tab.(!indice) <- tab.(!indice_fin); tab.(!indice_fin) <- buff; decr indice_fin
        end
    done;
    tab.(debut) <- tab.(!indice-1);
    tab.(!indice-1) <- pivot;
    !indice-1;;

let tri_rapide tab =
  let rec tri_rapide_aux debut fin =
    if debut < fin then
      begin
        let indice = split tab debut fin in
          tri_rapide_aux debut (indice-1);
          tri_rapide_aux (indice+1) fin
      end
  in tri_rapide_aux 0 (Array.length tab - 1);;
```

```
let rec perte cours liste = match liste with
| coursbis::q when cours.debut < coursbis.fin -> let p, reste = perte cours q in
  coursbis.satisfaction + p, reste
| _ -> 0, liste;;

let perte_minimale cours les_listes =
  let nbclones = Array.length les_listes in
  let indice = ref (-1) in
  let pertemin = ref cours.satisfaction in
  let resteindice = ref [] in
  for i = 0 to nbclones-1 do
    let pertetest, reste = perte cours les_listes.(i) in
    if pertetest < !pertemin then
    begin
      pertemin := pertetest;
      resteindice := reste;
      indice := i
    end
  done;
  !indice, !resteindice;;

let weighted_interval_scheduling tableau_cours nbclones =
  let cours = Array.copy tableau_cours in tri_rapide cours;
  let n = Array.length cours in
    let tableau_clones = Array.make nbclones [] in
    for i = 0 to n-1 do
      let indice, reste = perte_minimale cours.(i) tableau_clones in
      if indice <> -1 then
        tableau_clones.(indice) <- cours.(i)::reste
    done;
  tableau_clones;;

let satisfaction_maximale tableau_cours nbclones =
  let tableau_clones = weighted_interval_scheduling tableau_cours nbclones in
  let fonction1 accu cours = accu + cours.satisfaction in
  let fonction2 accu liste = accu + List.fold_left fonction1 0 liste in
  Array.fold_left fonction2 0 tableau_clones;;

let nombre_endroits_a_la_fois tableau_cours seuil =
  let n = Array.length tableau_cours in
  let rec essai i =
    if i > n then failwith "Jamais assez de satisfaction";
    if satisfaction_maximale tableau_cours i >= seuil then i (* et donc i-1 fois remonter le temps *)
    else essai (i+1)
  in essai 1;;
```

Quatrième année

Question P4.1 :

Pour arriver à la destination ou à tout endroit intermédiaire, il vaut mieux partir le plus tôt possible en raison de la propriété additionnelle donnée par l'énoncé. Il s'agit donc de stocker, comme dans le cas de l'algorithme de Dijkstra, l'information pour tout sommet du premier moment où il peut être rejoint. La seule différence à mettre en œuvre est d'adapter le typage pour que les arcs soient des fonctions, donc dans le programme de la partie cours, les deux mentions de `poids` deviennent `poids dist.(s)` et c'est tout.

Cinquième année

Question P5.1 :

Le travail était excessivement long, et mieux vaut en pratique faire le travail partiellement à chaque étape et utiliser la référence pour compléter (c'est l'esprit de LZW de pouvoir faire le travail sur un flux partiel, comme signalé en cours).

Séquence de départ :

```
44, 155, 206, 162, 3, 185, 164, 216, 108, 16, 24, 77, 135, 51, 120, 128, 204, 105, 55, 25, 4,
7, 67, 65, 132, 232, 32, 52, 25, 77, 135, 8, 28, 22, 15, 9, 59, 152, 79, 39, 49, 1, 136,
202, 32, 51, 154, 78, 198, 83, 116, 34, 44, 72, 55, 153, 228, 7, 35, 164, 140, 233, 12, 137,
155, 206, 114, 99, 185, 162, 24, 97, 57, 154, 225, 166, 243, 144, 128, 210, 116, 23, 0
```

Version en binaire avec tous les codes recollés :

```
00101100100110111100111010100010000000111011100110100100110110000011011000001000000001100001
00110110000111001100110111110001000000011001100011010010011011100011001000001000000001110100
00110100000110000100111010000010000000110100000110010100110110000011100001000000011100000101
10000011110000100100111011100110000100111100100111001100010000000110001000110010100010000
00110011100110100100111011000110010100110111010000100010001011000100100000110111100110011
10010000000111001000111010010010001100111010010000110010001001100110111100111001110010011 0
00111011100110100010000110000110000100111001100110101110000110100110111100111001000010000 0
00110100100111010000010111 0
```

Découpage en paquets de neuf bits (dernier paquet incomplet écarté) :

```
001011001, 001101111, 001110101, 000100000, 001110111, 001101001, 001101100,
001101100, 000100000, 001100001, 001101100, 001110011, 001101111, 000100000,
001100110, 001101001, 001101110, 001100100, 000100000, 001110100, 001101000,
001100001, 001110100, 000100000, 001101000, 001100101, 001101100, 001110000,
100000011, 100000101, 100000111, 100001001, 001110111, 001100001, 001111001,
001110011, 000100000, 001100010, 001100101, 000100000, 001100111, 001101001,
001110110, 001100101, 001101110, 100001000, 100010110, 001001000, 001101111,
001100111, 100100000, 001110010, 001110100, 100100011, 001110100, 100001100,
100010011, 001101111, 001110011, 100100110, 001110111, 001101000, 100001100,
001100001, 001110011, 001101011, 100001101, 001101111, 001110010, 000100000,
001101001, 001110100, 000101110
```

Entiers correspondants :

```
89, 111, 117, 32, 119, 105, 108, 108, 32, 97, 108, 115, 111, 32, 102, 105, 110, 100, 32, 116,
104, 97, 116, 32, 104, 101, 108, 112, 259, 261, 263, 265, 119, 97, 121, 115, 32, 98, 101, 32,
103, 105, 118, 101, 110, 264, 278, 72, 111, 103, 288, 114, 116, 291, 116, 268, 275, 111, 115,
294, 119, 104, 268, 97, 115, 107, 269, 111, 114, 32, 105, 116, 46
```

Et c'est enfin parti pour la décompression !

89 = Y, nouveau code en incorporant la première lettre du code 111 : 'Yo' (256). 111 = o, nouveau code : 'ou' (257). 117 = u, nouveau code : 'u ' (258).

Bref, après cette partie beaucoup plus agréable, on tombe sur la phrase *"You will also find that help will always be given at Hogwarts to those who ask for it."*.

Sixième année

Question P6.1 :

Matrice de départ :

Joueur	Gardien	Poursuiveur 1	Poursuiveur 2	Poursuiveur 3	Batteur 1	Batteur 2	Attrapeur
Demelza	1	3	3	3	1	1	5
Ginny	4	5	5	5	2	2	8
Harry	5	3	3	3	2	2	10
Jimmy	2	0	0	0	4	4	1
Katie	2	3	3	3	3	3	4
Ritchie	2	0	0	0	5	5	3
Ron	6	3	3	3	0	0	3

On retire le maximum de chaque ligne :

Joueur	Gardien	Poursuiveur 1	Poursuiveur 2	Poursuiveur 3	Batteur 1	Batteur 2	Attrapeur
Demelza	-4	-2	-2	-2	-4	-4	0
Ginny	-4	-3	-3	-3	-6	-6	0
Harry	-5	-7	-7	-7	-8	-8	0
Jimmy	-2	-4	-4	-4	0	0	-3
Katie	-2	-1	-1	-1	-1	-1	0
Ritchie	-3	-5	-5	-5	0	0	-2
Ron	0	-3	-3	-3	-6	-6	-3

On retire le maximum de chaque colonne (donc ici il faut augmenter des valeurs) :

Joueur	Gardien	Poursuiveur 1	Poursuiveur 2	Poursuiveur 3	Batteur 1	Batteur 2	Attrapeur
Demelza	-4	-1	-1	-1	-4	-4	0
Ginny	-4	-2	-2	-2	-6	-6	0
Harry	-5	-6	-6	-6	-8	-8	0
Jimmy	-2	-3	-3	-3	0	0	-3
Katie	-2	0	0	0	-1	-1	0
Ritchie	-3	-4	-4	-4	0	0	-2
Ron	0	-2	-2	-2	-6	-6	-3

Pour couvrir tous les zéros, on peut marquer les lignes Jimmy, Katie, Ritchie et Ron ainsi que la colonne Attrapeur.

Le maximum non marqué est alors -1, sur la ligne Demelza. On le soustrait donc (c'est-à-dire qu'on additionne 1) sur les trois premières lignes et on l'additionne sur la dernière colonne, conformément à l'énoncé. Le résultat figure ci-après.

Joueur	Gardien	Poursuiveur 1	Poursuiveur 2	Poursuiveur 3	Batteur 1	Batteur 2	Attrapeur
Demelza	-3	0	0	0	-3	-3	0
Ginny	-3	-1	-1	-1	-5	-5	0
Harry	-4	-5	-5	-5	-7	-7	0
Jimmy	-2	-3	-3	-3	0	0	-4
Katie	-2	0	0	0	-1	-1	-1
Ritchie	-3	-4	-4	-4	0	0	-3
Ron	0	-2	-2	-2	-6	-6	-4

On couvre désormais tous les zéros avec les mêmes lignes et colonnes marquées plus la ligne Demelza, et on répète avec le nouveau maximum qui est toujours -1 et cette fois-ci sur la ligne Ginny.

Joueur	Gardien	Poursuiveur 1	Poursuiveur 2	Poursuiveur 3	Batteur 1	Batteur 2	Attrapeur
Demelza	-3	0	0	0	-3	-3	-1
Ginny	-2	0	0	0	-4	-4	0
Harry	-3	-4	-4	-4	-6	-6	0
Jimmy	-2	-3	-3	-3	0	0	-5
Katie	-2	0	0	0	-1	-1	-2
Ritchie	-3	-4	-4	-4	0	0	-4
Ron	0	-2	-2	-2	-6	-6	-5

Il y a désormais une possibilité de sélectionner sept zéros de sorte qu'il y en ait exactement un par ligne et un par colonne. L'équipe type est donc : Ron gardien, Demelza, Ginny et Katie poursuiveuses, Jimmy et Ritchie batteurs et Harry attrapeur.

Septième année

Question P7.1 :

Vu que chaque année, chaque jour du mois existe selon chaque jour de la semaine (sauf éventuellement le 31), inutile de s'encombrer de la règle sur les années bissextiles pour les années séculaires, et même sur les années bissextiles tout court. On utilisera cette astuce dans la version proposée ci-après.

```
let nombre_jours = [| 31; 28; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31 |];;

let prochaine_fois day jour =
  let annee = ref 2024 and mois = ref 4 in (* décalé d'un pour commencer *)
  let jdls = ref ((2 + day) mod 7) in (* jour de la semaine du day du mois en cours *)
  let mois_ok = ref true in
  while !jdls <> jour || not !mois_ok do
    jdls := (!jdls + nombre_jours.(!mois)) mod 7;
    incr mois;
    if !mois = 12 then (mois := 0; incr annee);
    mois_ok := nombre_jours.(!mois) >= day
  done;
  (!annee, !mois+1);;
```

Question P7.2 :

On reprend l'algorithme de Rabin-Karp et on stocke un haché par chaîne à rechercher, rien de spécial à signaler.

Pour la fonction de hachage de l'exercice, et afin de ne pas avoir trop de paramètres, on choisit arbitrairement les valeurs `b = 97` et `p = 43`.

```
int hachage(char* s, int taille) // Une fonction suffira cette fois
{
  int final = 0;
  for (int i = 0 ; i < taille ; i += 1)
  {
    final = final * 97 + (int) s[i];
    final = final % 43;
  }
  return final;
}

int hachage_suivant (int x, char* s, int k, int bpkm1, int ind)
{
  return (x - s[ind] * bpkm1) * 97 + s[ind + k];
}
```

```
int rabin_karp_motif_plus_frequent(char* s, char** m, int n) {
  assert (n > 0);
  int ns = strlen(s);
  int nm = strlen(m[0]);
  if (nm > ns) return 0;
  int* nb_occ = malloc(n * sizeof(int));
  int* haches = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1)
  {
    nb_occ[i] = 0;
    haches[i] = hachage(m[i], nm);
  }
  int hs = hachage(s, nm);
  int bpkm1 = 1;
  for (int i = 1 ; i < nm ; i += 1)
  {
    bpkm1 = bpkm1 * 97 % 43;
  }
  for (int i = 0 ; i < ns - nm ; i++)
  {
    for (int j = 0 ; j < n ; j += 1)
    {
      if (haches[j] == hs)
      {
        int count = 0;
        while (count < nm && s[i + count] == m[j][count]) count += 1;
        if (count == nm) nb_occ[j] += 1;
      }
    }
    hs = hachage_suivant(hs, s, nm, bpkm1, i) % 43;
  }
  int rep = 0;
  for (int j = 1 ; j < n ; j += 1)
  {
    if (nb_occ[j] > nb_occ[rep]) rep = j;
  }
  free(nb_occ);
  free(haches);
  return rep;
}
```

Question P7.3 :

Voici l'algorithme en temps linéaire et en espace constant : on considère successivement tous les indices `i` dans le tableau `t` et on étudie pour commencer si `t[i]` est égal à `i`. Le nombre de fois où ceci n'est pas réalisé sera la taille du cycle s'il est unique. En parallèle, on peut mémoriser un booléen déterminant si on a déjà emprunté un cycle pour ne le faire qu'une fois (limiter la complexité), et quand on emprunte le cycle on mémorise l'indice de début et la taille. À la fin, on vérifie la cohérence de toutes les informations récupérées.

Pour éviter une boucle infinie si le tableau ne respecte pas les conditions, on pourra déclencher une erreur si la taille du cycle dépasse la taille du tableau. Ceci donnera un variant pour prouver la terminaison de la boucle conditionnelle explorant le cycle.

La complexité sera bien celle qu'on a annoncée (chaque élément ne sera considéré qu'une fois pour déterminer s'il est un point fixe de la permutation, et en plus de cela on explore un cycle de taille linéaire).

Implémentation en OCaml :

```
let un_cycle tab =
  let n = Array.length tab in
  let nb_fixes = ref 0 in
  let deja_cycle = ref false in
  let taille_cycle = ref 0 in
  for i = 0 to n-1 do
    if tab.(i) = i then incr nb_fixes
    else if not !deja_cycle then
    begin
      deja_cycle := true;
      let position = ref tab.(i) in
      incr taille_cycle;
      while !taille_cycle <= n && !position <> i do
        incr taille_cycle;
        position := tab.(!position)
      done;
      if !taille_cycle > n then failwith "Mauvais format"
    end
  done;
  !nb_fixes + !taille_cycle = n;;
```

Question P7.4 :

Il n'y avait pas d'objectif de complexité ici. L'algorithme que l'on propose ici revient à faire de l'exploration exhaustive.

En premier, on vérifie si la réponse n'est pas zéro : on recense les sommets dont la différence entre degré sortant et degré entrant n'est pas nulle. S'il n'y en a aucun, on lancera l'exploration exhaustive depuis chaque sommet et on additionnera les résultats (les chemins obtenus depuis chaque sommet sont forcément différents d'un sommet à l'autre), s'il y en a deux et que la différence est d'un en valeur absolue, on ne lance l'exploration exhaustive que depuis le sommet qui a un degré sortant égal à son degré entrant plus un (et si le graphe n'est pas « connexe » au sens intuitif que l'on peut donnner à la connexité d'un graphe orienté, on échouera à trouver le moindre chemin eulérien).

L'exploration exhaustive depuis un sommet revient à lancer un parcours mais en considérant les arcs comme fermés plutôt que les sommets, de sorte qu'un sommet puisse être visité plusieurs fois. Une fois que le nombre d'arcs dans le chemin en construction est égal au nombre total d'arcs, on compte un chemin eulérien de plus et on procède à un retour sur trace, alors que si on atteint un sommet dont plus aucun arc sortant n'est disponible sans que la taille du chemin ne soit suffisante, on procède à un retour sur trace sans comptabiliser le chemin.

La terminaison se prouve de la même manière qu'on prouve la terminaison d'un parcours.

La complexité est ici au-delà d'exponentielle, car le nombre de chemin eulériens l'est, ce qui nous amène à donner une autre méthode de résolution, par le théorème de BEST (du nom des auteurs) donnant une formule pour le nombre de circuits eulériens dans un graphe orienté « connexe » (même remarque) dont chaque sommet est de degré entrant égal au degré sortant, cette formule faisant intervenir le produit des factorielles des degrés sortants moins un.

Question P7.5 :

Un algorithme de calcul des cliques maximales a été publié par Coenraad Bron et Joep Kerbosch en 1973. Le nombre de cliques maximales peut être exponentiel en le nombre de sommets du graphe, ce qui donne une borne inférieure particulièrement élevée pour la complexité d'un algorithme résolvant le problème.

L'algorithme ci-avant procède à une exploration exhaustive astucieuse. On peut improviser un algorithme similaire en commençant par trouver tous les candidats à être des cliques maximales : les sommets isolés (dont on ne reparlera plus) et les sous-ensembles de deux sommets reliés par une arête. Par la suite, on peut tester chacun des ensembles actuellement candidats et pour chaque sommet n'y étant pas, on regarde si l'ajout garde la propriété de clique (et qu'on n'a pas encore cette clique dans la liste des candidats). Si aucun sommet ne peut s'ajouter, la clique est maximale et on ne reviendra plus dessus non plus. Tous les autres nouveaux candidats sont ajoutés à la séquence en attente (une file, par exemple).

(Pour faciliter le travail, on peut mémoriser en association aux cliques candidates quels sommets ne peuvent pas s'ajouter pour ne plus les considérer au moment des tests, c'est ce que proposaient les auteurs mentionnés ci-avant.)

Bien entendu, une fois la liste des cliques obtenue, une quadruple boucle (pour toute clique, pour toute autre clique, explorer les sommets de l'un, tester l'égalité avec un sommet de l'autre) permet de trouver les arcs du graphe des cliques. En admettant une structure efficace pour la représentation des cliques (tableau de booléens ou dictionnaire), on aura juste un temps quadratique en le nombre de cliques, multiplié par le nombre de sommets, pour la construction du graphe. On n'est plus à cela près.

Question P7.6 :

On commence par reprendre l'algorithme de Boyer-Moore en construisant une table de redirection. Celle-ci sera valable pour tous les cas où on cherchera un facteur démarrant à une position supérieure à la position de départ. Ceci étant, il faudra aussi chercher les positions inférieures à la position de départ, et pour celles-ci on doit utiliser une version « miroir », donc on comparera les caractères de la gauche vers la droite pour décaler plus rapidement l'indice de départ vers la gauche en moyenne. On lancera alors la recherche vers la droite et la recherche vers la gauche soit l'une après l'autre et on comparera le premier résultat de chaque recherche, soit en une fois à condition de mémoriser les positions actuelles et de considérer à chaque fois celle qui est la plus proche de la position de départ en priorité.

Question P7.7 :

Rappel de la séquence en hexadécimal :

```
24 05 8E E0 B7 5C CB 7A 92 97 5B 16 52
D1 61 BA 51 17 31 DF 3F 98 4B C7 D6 02
```

Les entiers correspondants :

```
36, 5, 142, 224, 183, 92, 203, 122, 146, 151, 91, 22, 82,
209, 97, 186, 81, 23, 49, 223, 63, 152, 75, 199, 214, 2
```

La séquence de bits associée :

```
0010010000000101
1000111011100000
1011011101011100
1100101101111010
1001001010010111
0101101100010110
0101001011010001
0110000110111010
0101000100010111
0011000111011111
0011111110011000
0100101111000111
1101011000000010
```

Découpage pour faire apparaître l'arborescence :

```
0
 0
  1 00100000 [espace, code 00]
  0
   0
    1 01100011 [c, code 0100]
    1 01110000 [p, code 0101]
   0
    1 01101110 [n, code 0110]
    1 01110011 [s, code 0111]
 0
  0
   1 01101111 [o, code 100]
   0
    1 01001001 [I, code 1010]
    0
     1 00101110 [point, code 10110]
     1 01101100 [l, code 10111]
  0
   1 01100101 [e, code 110]
   0
    0
     1 01101000 [h, code 11100]
     1 01100001 [a, code 11101]
    1 01110100 [t, code 1111]
10100010001011100110001110111110011111110011000010010111100011111010110000000010
```

Au vu des trois derniers bits, on ne considère que huit bits des deux derniers octets, donc cinq zéros sont exclus à la fin.

Les codes découpés selon l'arborescence :

```
1010
00
100
0101
110
0110
00
11101
1111
00
1111
11100
110
00
0100
10111
100
0111
110
10110
```

Et la phrase est « *I open at the close.* » mais on aurait pu s'y attendre vu le contexte.