Corrigé de l'épreuve d'informatique.
Concours Communs Polytechniques 97.

PARTIE I : épreuve de logique.

1. **A** et **B** s'écrivent respectivement $(\mathbf{OD} \vee \mathbf{OG})$ et $(\neg\mathbf{OD})$.

2. La connaissance de **C** permet donc de dire que la formule

$$[(\mathbf{OD} \vee \mathbf{OG}) \wedge (\neg\mathbf{OD})] \vee [\neg(\mathbf{OD} \vee \mathbf{OG}) \wedge (\mathbf{OD})]$$

est vraie.

3. La proposition $\neg(\mathbf{OD} \vee \mathbf{OG}) \wedge (\mathbf{OD})$ est équivalente à $(\neg\mathbf{OD}) \wedge (\neg\mathbf{OG}) \wedge \mathbf{OD}$. C'est donc une antilogie. D'après les formules de Morgan, la formule propositionnelle $(\mathbf{OD} \vee \mathbf{OG}) \wedge (\neg\mathbf{OD})$ est équivalente à la formule propositionnelle $(\mathbf{OD} \wedge \neg\mathbf{OD}) \vee (\mathbf{OG} \wedge \neg\mathbf{OD})$. Comme $(\mathbf{OD} \wedge \neg\mathbf{OD})$ est une antilogie, cette dernière formule est équivalente à $(\mathbf{OG} \wedge \neg\mathbf{OD})$. Nous en déduisons donc que **OG** est vraie et que **OD** est fausse. Ainsi la piste de gauche (et elle seule) conduit à une oasis.

3.bis On remarquera tout de même que cette partie ferait un bon exercice pour le concours Kangourou niveau collège, la solution se trouvant "de tête" : si les deux sphinx mentent, nous obtenons une absurdité puisqu'aucune piste ne mène à une oasis et que la piste de droite conduit à une oasis ! Les deux sphinx disent donc la vérité, c'est à dire que la piste de droite se perd dans le désert et que la piste de gauche conduit à une oasis.

PARTIE II : exercice sur les automates finis.

1.a) Le langage reconnu par A est $L = (a^+b^2)^*a^+b$. C'est l'ensemble des mots de la forme

$$a^{n_1}b^2a^{n_2}b^2 \dots a^{n_{k-1}}b^2a^{n_k}b,$$

avec $k \geq 1$ et $n_1, \dots, n_k \geq 1$.

1.b) ab est élément de L et ε est un préfixe de ab. Nous en déduisons que $q_0 = \delta_A^*(q_0, \varepsilon)$ est élément de G.

La suite (G_i) est une suite croissante de parties de $\{q_0, q_1, q_2, q_3\}$. Elle est donc stationnaire. Il existe ainsi un rang j tel que $G_j = G_{j+1}$. La définition correcte des G_i est sans doute quelque chose comme :

$$G_{i+1} = G_i \cup \{q \in \mathbf{G} | \exists q' \in G_i, \exists x \in X : q = \delta_A(q', x)\}.$$

Nous avons alors par définition $G_j \subset G$. Réciproquement, si q est élément de G, il existe w et y dans X^* tels que $q = \delta_A^*(q_0, w)$ et $wy \in L$. Si nous notons $w = x_0x_1 \dots x_{k-1}$ avec $k \in \mathbb{N}$ et $x_0, \dots, x_{k-1} \in X$, nous avons $q_0 \in G_0$, $q_1 = \delta_A(q_0, x_0) \in G_1$, $q_2 = \delta_A(q_1, x_1) \in G_2$, $\dots$, $q_k = \delta_A(q_{k-1}, x_{k-1}) \in G_k$. Mais $q_k = q$, et donc $q \in G_k$. Comme $G_k \subset G_j$ (on montre facilement que la suite (G_i) est stationnaire à partir du rang j), nous avons bien $q \in G_j$.

1.c) L'automate B est décrit par la figure 1. Nous comprendrons la question de la détermination de l'automate B comme consistant simplement à trouver un automate déterministe reconnaissant le même langage que B(*). Il suffit ici (et cela fonctionne dans le cas général) de considérer l'automate C défini par :

$$C = (Q_A, X, \delta_A, q_0, G).$$

(*) On pourrait imaginer que l'énoncé nous demande d'appliquer le processus général de déterminisation d'un automate asynchrone au cas particulier traité ici, mais la connaissance de cette technique n'est pas exigée par le programme du cours d'informatique.

Cet automate est obtenu en changeant l'ensemble des états terminaux de l'automate A : les nouveaux états terminaux sont simplement les états par lesquels passent les différents chemins réussis de l'automate A. Cette remarque montre que le résultat démontré dans cet exercice est presque trivial, et que l'introduction des ε-transitions est inutile ! Dans l'exemple traité, l'automate obtenu ainsi est schématisé par la figure 2.

2. Le résultat est pratiquement évident :

• si w est reconnu par B, le mot w est l'étiquette d'un chemin allant de q_0 à t dans B. Il existe donc $q \in G$ tel que $\delta_A^*(q_0, w) = q$. Mais comme q est élément de G, il existe $y \in X^*$ tel que $\delta_A^*(q, y) \in F_A$. Ainsi $\delta_A^*(q_0, wy) \in F_A$, et donc w est un préfixe d'un mot reconnu par A.

• soit $w \in \text{Préf}(L)$. Fixons $y \in X^*$ tel que $wy \in L$. Alors $q = \delta_A^*(q_0, w)$ est un élément de G, et donc w est un mot reconnu par l'automate B.

PARTIE III : épreuve d'algorithmique.

En langage Caml

I. Calcul des parties d'un ensemble fini.

1. Nous avons clairement $\mathcal{P}(\{x\} \cup e) = \mathcal{P}(e) \cup x \bullet \mathcal{P}(e)$.

2. Notons $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) l'ensemble des parties de $\{x\} \cup e$ contenant x (resp. ne contenant pas x). Alors l'application qui, à un élément e' de $\mathcal{P}_2$, associe $\{x\} \cup e'$ réalise une bijection de $\mathcal{P}_2$ sur $\mathcal{P}_1$. Ces deux ensembles ont donc même cardinal.

3. Pour chaque entier naturel n, notons a_n la somme des cardinaux des parties d'un ensemble à n éléments. Si n est un entier au moins égal à 2 et si E est un ensemble à n éléments, dont nous distinguons l'élément x, nous pouvons appliquer la question précédente à la partie $e = E \setminus \{x\}$: l'ensemble $\mathcal{P}_1$ des parties de E contenant x et l'ensemble $\mathcal{P}_2$ des parties ne contenant pas x forment une partition de $\mathcal{P}(E)$ en deux parties de même cardinal. Ainsi, $\mathcal{P}_1$ et $\mathcal{P}_2$ ont 2^{n-1} éléments. Nous avons alors :

$$\begin{aligned} a_n &= \sum_{f \subset E} |f| = \sum_{f \in \mathcal{P}_2} |f| + \sum_{f \in \mathcal{P}_2} |f \cup \{x\}| = 2 \sum_{f \in \mathcal{P}_2} |f| + \sum_{f \in \mathcal{P}_2} 1 \\ &= 2a_{n-1} + |\mathcal{P}_2| = 2a_{n-1} + 2^{n-1}. \end{aligned}$$

On montre alors facilement que la suite (b_n) définie par $b_n = n2^{n-1}$ pour $n \geq 1$ vérifie également la relation de récurrence

$$b_n = 2b_{n-1} + 2^{n-1}$$

pour tout $n \geq 2$. Comme $a_1 = b_1 = 1$, on en déduit que les suites (a_n) et (b_n) sont égales.

On peut remarquer qu'il est plus facile d'oublier la question 2. et d'écrire :

$$a_n = \sum_{k=0}^{n} k\, C_n^k = n2^{n-1},$$

relation obtenue classiquement en dérivant l'identité $(1 + x)^n = \sum_{k=0}^{n} C_n^k x^k$.

4. et **5.** Nous obtenons sans problème les procédures **ajouter** et **parties**.

```
let rec ajouter x e =
match e with
   [ ] -> [ ]
   | t :: f -> (x :: t) :: ajouter x f ; ;
```

```
let rec parties = function
    [ ] -> [[ ]]
    | t::x -> let u = parties x in (ajouter t u) @ u;;
```

II. Application au problème dit du sac-à-dos.

A.1. Nous utiliserons les trois fonctions **fst**, **snd** et **thrd** qui renvoient respectivement le premier, le deuxième et le troisième argument d'un triplet auquel on les applique.

```
let fst(x,y,z) = x ;;        let snd(x,y,z) = y ;;        let thrd(x,y,z) = z ;;
```

On adapte facilement la fonction **ajouter** comme proposé par l'énoncé.

```
let rec ajouter_objet x e =
match e with
    [ ] ->  [ ]
    | tete :: queue -> (x :: fst(tete), p(x) + snd(tete), v(x) + thrd(tete)) :: ajouter_objet x queue;;
```

```
let rec chargement = function
    [ ] -> [ ]
    | t::x -> let u = chargement x in ([t],p(t),v(t))::u @ (ajouter_objet t u) ;;
```

A.2. Nous donnons tout d'abord le code de la fonction **duel**. Celle-ci prend en arguments P,V,e1 et e2 où e1 et e2 sont des triplets de type ('a list * int * int), et renvoie le meilleur des deux chargements e1 et e2. L'algorithme utilisé sous-entend que e2 est un chargement "correct", c'est à dire que **snd**(e2) et **thrd**(e2) sont plus petits respectivement que P et V(*).

```
let duel P V e1 e2 =
(if (thrd(e1)>thrd(e2) & thrd(e1)<=V & snd(e1)<=P) or (thrd(e1)=thrd(e2) & snd(e1)>snd(e2) &
snd(e1)<=P) then
    e1
else
    e2) ;;
```

Nous obtenons alors la fonction **sac_a_dos** en appliquant récursivement la fonction locale **optimal** à la liste calculée par la fonction **chargement**.

```
let sac_a_dos P V e =
    (let rec optimal = function
        [ ] ->  ([ ], 0, 0)
        | t :: x -> duel P V t (optimal x)
    in optimal(chargement(e))) ;;
```

B.1. Nous obtenons directement :

$u_1 = [1]$ $u_2 = [2;1]$ $u_3 = [3;2;1]$ $u_4 = [3;1]$
$u_5 = [2]$ $u_6 = [3;2]$ $u_7 = [3]$

B.2. Il suffit de distinguer suivant que u est de la forme x::y::t ou [x] pour écrire la fonction **terme_suivant**. Quand u=[n] (u code alors la dernière des parties non vides de $1..n$) , la fonction renvoie la liste vide. Quand u=[ ], la fonction renvoie la suite [1].

(*) L'énoncé est à ce propos ambigu : il est possible de comprendre que la condition "poids $\leq$ P" n'est prise en compte que lorsque deux chargements correspondent au volume transportable maximal.

```
let terme_suivant n u =
match u with
    x :: y :: t -> (if x=n then y+1 :: t else x+1 :: u)
  | [x] -> (if x=n then [ ] else x+1 :: u)
  | [ ] -> [1] ; ;
```

B.3. La fonction **sac_a_dos'** reprend la structure de la fonction **enumerer** en stockant dans la variable optimal la meilleure solution parmi les chargements étudiés. L'énoncé demande d'utiliser la fonction **terme_suivant**, mais il aurait été intéressant de modifier cette fonction pour lui faire calculer simultanément $u_p = [y_1, \ldots, y_l]$, $P_p = p(y_1) + \cdots + p(y_l)$ et $V_p = v(y_1) + \cdots + v(y_l)$ en fonction de n, $u_{p-1} = [x_1, \ldots, x_k]$, $P_{p-1} = p(x_1) + \cdots + p(x_k)$ et $V_{p-1} = v(x_1) + \cdots + v(x_k)$. Nous utiliserons les fonctions auxilliaires **poids** et **volume** qui, appliquées à une liste $[x_1; x_2; \ldots; x_k]$ renvoient respectivement $p(x_1) + \cdots + p(x_k)$ et $v(x_1) + \cdots + v(x_k)$.

```
let rec poids = function
    [ ] -> 0
  | x :: u ->  p(x) + poids(u) ; ;

let rec volume = function
    [ ] -> 0
  | x :: u ->  v(x) + volume(u) ; ;

let sac_a_dos' P V n =
    let n1 = deux_puissance n and opt = ref([1],p(1),v(1)) and u = ref[1] in
        for i=1 to n1-2 do
            u := terme_suivant n !u ;
            opt :=duel P V ( !u,poids !u,volume !u) !opt
        done ;
!opt ; ;
```

C.1. La fonction **terme_suivant** appliquée à n et u_p (pour p compris entre 1 et $2^n - 1$) utilise une fois l'opérateur **: :**. Nous avons donc 2 appels au constructeur **: :** dans chaque boucle de la fonction **enumerer**, et donc $T(n) = 2 \times (n1 - 2) = 2 \times 2^n - 4$.

C.2. Notons $T_a(n)$ la complexité de le fonction **ajouter t u**, où n est la longueur de la liste u. Nous avons $T_a(0) = 0$ et $T_a(n+1) = T_a(n) + 2$, d'où $T_a(n) = 2n$. Si $T_p(n)$ désigne la complexité de la fonction **parties**, nous avons : $T_p(0) = 0$ et $T_p(n+1) = T_p(n) + 2 \times 2^n + 2^n$. En effet, la liste e est de la forme t :: x avec x de longueur n. Ainsi, la liste u=**parties** x est de longueur 2^n. Le calcul de **ajouter** t u prend donc un temps égal à 2×2^n, et la concaténation avec la liste u prend un temps égal à la longueur de **ajouter** t u, qui est encore égale à 2^n. Nous en déduisons donc la relation

$$T_p(n) = 3 \times (2^n - 1).$$

C.3. La fonction **enumerer** calcule la suite des parties de $1..n$ en un temps équivalent à 2×2^n, alors que la fonction **parties** demande un temps de calcul équivalent à 3×2^n, qui est 50% supérieur.