Tableau de bord (d'après le concours Polytechnique)

2 Parties de $\mathbb{N}$

1. Le plus simple est sans doute de décomposer le test. On peut commencer par écrire une fonction testant si une liste est strictement croissante (ce qui impose par ailleurs des éléments distincts) :

```
let rec increasing = function
  | h1::h2::t -> h1 < h2 && increasing (h2::t)
  | _ -> true
```

Puis garantir qu'ils sont positifs en vérifiant la tête, si elle existe (on pourrait bien évidemment combiner ces deux fonctions en une seule) :

```
let valid lst =
  lst = [] || List.hd lst >= 0 && increasing lst
```

Attention au style, même si « `if XXX then true else false` » n'est pas incorrect, « `XXX` » étant un booléen, on peut l'utiliser directement et se passer du test! De même, « `if XXX then YYY else false` » peut s'écrire « `XXX && YYY` ». La concision n'est pas un objectif en soi, mais elle conduit souvent à une fonction plus aisée à lire, ce qui doit en revanche être une préoccupation lorsque l'on écrit du code.

2. On peut distinguer les différents cas de figure :

```
let rec delta p1 p2 = match p1, p2 with
  | _, [] -> p1
  | [], _ -> p2
  | h1::t1, h2::t2 when h1<h2 -> h1::delta t1 p2
  | h1::t1, h2::t2 when h1>h2 -> h2::delta p1 t2
  | h1::t1, h2::t2 -> delta t1 t2
```

3 Énumération des parties par incrément

3. Pour s'aider, il ne faut pas hésiter à considérer des exemples (nombreux) : `[]` donne `[0]`, puis `[1]`, puis `[0; 1]`, puis `[2]`, puis `[0; 2]`, puis `[1; 2]`, puis `0; 1; 2]`, puis `[3]`, puis `[0; 3]` (il s'agit d'une décomposition des entiers de 0 à 9). `[1; 3; 5]` (42) donne `[0; 1; 3; 5]`, puis `[2; 3; 5]`, puis `[0; 2; 3; 5]`...

Il s'agit en fait d'ajouter le plus petit entier n'apparaissant pas dans la partie et de retirer tous les entiers plus petits présents. On peut y parvenir avec une fonction auxiliaire qui prend en argument un entier `i` et une partie constituée d'entiers supérieurs ou égaux à `i`, et qui ajoute `i` à la partie si `i` n'y figure pas (il serait en tête par construction), et sinon tente d'ajouter `i+1` à la partie privée de `i` :

```
let succ lst =
  let rec aux i lst = match lst with
    | h::t when i=h -> aux (i+1) t
    | _ -> i::lst
  in aux 0 lst
```

Une autre solution pertinente que j'ai vue dans certaines copies consiste à ajouter un 0 en tête de liste (ce qui ajoute bien $2^0 = 1$ au total associé), et tant que la liste commence par deux éléments égaux h, on les remplace tous les deux par un unique $h+1$:

```
let succ lst =
  let rec aux = function
    | h1::h2::t when h1=h2 -> aux (h1+1::t)
    | lst -> lst
  in aux (0::lst)
```

4. On peut énumérer les parties avec succ jusqu'à parvenir à une partie réduite à N (la première faisant intervenir un entier supérieur ou égal à N, et la 2^{N_e} partie incidemment). Pour obtenir les interrupteurs à basculer, on utilise `delta` (excepté pour la dernière étape où on rebascule tous les interrupteurs).

```
let sequence n =
  let rec aux p = match succ p with
    | [i] when i=n -> [p]
    | p' -> delta p p'::aux p'
  in aux []
```

Il existe quantité d'autres façon d'obtenir la liste demandée. Par exemple, on peut remarquer que 0 apparaît chaque fois, 1 une fois sur deux, 2 une fois sur quatre, etc. Et remarquer que k apparaît à la position i dans la liste si et seulement si il y a au moins k zéros à droite de l'écriture binaire de $i+1$. On peut construire une liste des entiers de 0 à k où k est le nombre de zéros à droite dans l'écriture binaire d'un entier $n > 0$ en écrivant :

```
let build n =
  let rec aux i n =
    if n mod 2 = 0 then i::aux (i+1) (n/2) else [i]
  in aux 0 n
```

On définit également une fonction permettant de déterminer 2^n :

```
let rec pow2 n =
  if n=0 then 1 else 2*pow2 (n-1)
```

On serait alors tenté d'écrire

```
let sequence n =
  List.init (pow2 n) (fun i -> build (i+1)) (* faux ! *)
```

Mais il y a un petit souci : N apparaîtrait en dernière place de la dernière liste, ce que l'on ne souhaite pas. Plutôt que de l'enlever, on peut remarquer que la première moitié de la liste demandée est identique à la seconde, et écrire plutôt :

```
let sequence n =
  let lst = List.init (pow2 (n-1)) (fun i -> build (i+1))
  in lst @ lst
```

Cela dit, si l'on remarque que la seconde moitié de la liste est identique à la première, on peut aussi remarquer que la seconde moitié de la première moitié est identique à la première moitié de la première moitié, au seul ajout près d'un $n-1$ en fin de la toute dernière liste. Et récursivement. Pour des raisons de simplicité, on peut construire la liste retournée des listes retournées, et opérer tous les retournements à la fin. Ce qui donne la fonction ci-dessous, *qui devrait être davantage commentée*, mais que je vous laisse analyser.

```
let sequence n =
  let rec aux = function
    | 0 -> [[0]]
    | n -> let lst = aux (n-1) in (n::List.hd lst)::List.tl lst @ lst
  in let lst = aux (n-1) in List.rev (List.map List.rev (lst @ lst))
```

Comme vous le voyez, il y a des dizaines de solutions différentes, **mais elles nécessitent d'être expliquées**, d'autant qu'il est aisé de commettre une petite erreur dans la construction.

5. La discussion précédente contient la réponse à cette question : l'interrupteur 0 est commuté 2^N fois, le 1 l'est 2^{N-1} fois, jusqu'au $N-1$ qui l'est deux fois.

Au total, $2^N + 2^{N-1} + 2^{N-2} + \ldots + 2 = 2^{N+1} - 2$ commutations.

4 Énumération des parties par un code de Gray

6. $T(4) = [0,1,0,2,0,1,0,3,0,1,0,2,0,1,0]$.

7. $S_0 = \emptyset$, $S_1 = \{0\}$, $S_2 = \{0,1\}$, $S_3 = \{1\}$, $S_4 = \{1,2\}$, $S_5 = \{0,1,2\}$, $S_6 = \{0,2\}$, $S_7 = \{2\}$, $S_8 = \{2,3\}$, $S_9 = \{0,2,3\}$, $S_{10} = \{0,1,2,3\}$, $S_{11} = \{1,2,3\}$, $S_{12} = \{1,3\}$, $S_{13} = \{0,1,3\}$, $S_{14} = \{0,3\}$, $S_{15} = \{3\}$

8. Pour parvenir à S_{2^n-1}, si $n > 0$, on commute un nombre pair de fois tous les interrupteurs inférieurs ou égaux à $n-2$ (du fait de la double apparition de $T(n)$ dans l'expression) et une seule fois l'interrupteur $n-1$, donc $S_{2^n-1} = \{n-1\}$.

Dans le cas particulier $n = 0$, on a $S_{2^0-1} = S_0 = \emptyset$.

9. On a $S_{2^n} = S_{2^n-1} \,\Delta\, t_{2^n} = \{n-1\} \,\Delta\, \{n\} = \{n, n-1\}$.

Ensuite, on remarquera que $t_{2^n+i} = t_i$ par construction de T (à cause de la répétition de $T(n)$ dans la définition, et en remarquant que la longueur de T_n est $2^n - 1$) pour tout $i < 2^n - 1$. On peut donc montrer que pour tout $i < 2^n$, $S_{2^n+i} = S_i \,\Delta\, \{n, n-1\}$ par récurrence sur i :

- c'est vrai pour $i = 0$, puisque $S_0 = \emptyset$ et $S_{2^n} = \{n, n-1\}$;
- si $S_{2^n+i} = S_i \,\Delta\, \{n, n-1\}$ et $i < 2^n - 1$, alors $S_{2^n+i+1} = S_{2^n+i} \,\Delta\, t_{2^n+i} = (S_i \,\Delta\, \{n, n-1\}) \,\Delta\, t_{2^n+i} = (S_i \,\Delta\, \{n, n-1\}) \,\Delta\, t_i = (S_i \,\Delta\, t_i) \,\Delta\, \{n, n-1\} = S_{i+1} \,\Delta\, \{n, n-1\}$ (respectivement définition de S_k, hypothèse de récurrence, remarque précédente sur t_{2^n+i}, associativité et commutativité de Δ).

Par récurrence, on a donc bien pour tout $i < 2^n$, $S_{2^n+i} = S_i \,\Delta\, \{n, n-1\}$.

10. On peut le faire par récurrence sur n.

H_n : $P_n = \{S_0, S_1, \ldots, S_{2^n-1}\}$ est l'ensemble des parties de I_n.

C'est vrai pour H_1, H_2, H_3 et H_4 comme on peut le voir ci-dessus.

Supposons H_n vrai, et étudions H_{n+1}. Pour montrer qu'il s'agit des parties de I_{n+1}, il suffit demontrer que tout les S_k sont distincts puisque le cardinal est celui attendu. La première moitié des S_k représente, par réccurrence, les parties de I_n, donc ils sont tous distincts et n n'y apparaît pas. Pour les 2^n restants, d'après la question précédente, n apparaît dans chacun d'entre eux.

Il ne reste donc qu'à prouver que les 2^n parties de la seconde moitié des S_k sont distincts deux à deux. Mais s'il existe k et k' différents tels que $S_{2^n+k} = S_{2^n+k'}$, alors $S_{2^n+k} \,\Delta\, \{n-1, n\} = S_{2^n+k'} \,\Delta\, \{n-1, n\}$, donc $S_k = S_{k'}$, ce qui est impossible.

Tous les éléments sont donc bien distincts deux à deux, et H_n est vérifié.

11. On remarquera que si le sujet laisse une certaine ambiguité sur la question, chacune des 2^N listes d'indices est une liste à un seul élément ! Il s'agit par ailleurs pratiquement des t_i, à une seule exception : le dernier qui ne doit pas être N mais $N-1$. Par exemple, pour $N = 3$, on veut

```
[[0]; [1]; [0]; [2]; [0]; [1]; [0]; [2]]
```

Là encore, il existe *énormément* de façons d'obtenir cette liste. Une amusante (mais ineffi-

cace) consisterait simplement à prendre le dernier élément de chaque liste de sequence n :

```
let sequence_gray n =
  List.map (fun lst -> [List.hd (List.rev lst)]) (sequence n)
```

Plus efficacement, on peut simplement construire $T(n)$ (attention, on ne peut pas utiliser de majuscule pour les noms en OCaml, on écrit donc « t ») :

```
let rec t = function
  | 0 -> []
  | n -> let lst = t (n-1) in lst @ (n-1) :: lst
```

Et ajouter un $n-1$ à la fin, et on place chaque élément seul dans une liste :

```
let sequence_gray n =
  List.map (fun x -> [x]) (t n @ [n-1])
```

Même si le sujet n'impose pas de complexité, on peut vérifier que malgré la présence de concaténations, la fonction reste de complexité linéaire en la longueur de la liste, 2^N.

12. Comme chaque liste est de longueur 1, et qu'il y en a 2^N, il y a donc exactement 2^N interrupteurs à commuter. Et comme il y a 2^N configurations distinctes à visiter, et qu'il faut au moins commuter un interrupteur pour passer de l'une à l'autre, il faut au moins 2^N-1 commutations pour les visiter toutes, et une dernière pour revenir à la situation initiale. On ne peut pas faire mieux.

13. Montrons que $t_i = \min(S_i)+1$ lorsque i est impair.

Pour ce faire, utilisons une récurrence sur n pour montrer que pour tout i impair dans $[\![2^n \,..\, 2^{n+1}-1]\!]$ on a $t_i = \min(S_i)+1$.

C'est vrai pour $n=0$ (soit $i=1$) car $t_1 = 1$ et $S_1 = \{0\}$.

Supposons cela vrai pour n. Soit i impair dans $[\![2^{n+1} \,..\, 2^{n+2}-1]\!]$.

Si $i = 2^{n+2}-1$, d'après les questions précédentes, $S_i = \{n+1\}$ et $t_i = n+2$, donc cela convient.

Sinon, $i < 2^{n+2}-1$, $S_i = S_{i-2^{n+1}} \,\Delta\, \{n, n+1\}$. Or $i-2^{n+1}$ est impair et $0 < i-2^{n+1} < 2^{n+1}$. Donc par récurrence, $t_{i-2^{n+1}} = 1+\min\left(S_{i-2^{n+1}}\right)$. Et on sait que par construction, $t_i = t_{i-2^{n+1}}$.

Il ne reste donc qu'à montrer que $\min\left(S_{i-2^{n+1}}\right) = \min(S_i)$, en sachant que $S_i = S_{i-2^{n+1}} \,\Delta\, \{n+1, n\}$. Or S_i ne peut être parmi $\emptyset$, $\{n+1\}$, $\{n\}$ et $\{n, n+1\}$ puisque ces configurations sont S_0, $S_{2^{n+2}-1}$, $S_{2^{n+1}-1}$ et $S_{2^{n+1}}$. Par conséquent, il y a nécessairement un entier strictement inférieur à n dans S_i et $S_{i-2^{n+1}}$, donc on a bien $\min\left(S_{i-2^{n+1}}\right) = \min(S_i)$.

Par récurrence, pour tout n, pour tout i impair dans $[\![2^n \,..\, 2^{n+1}-1]\!]$ on a $t_i = \min(S_i)+1$, donc la relation est vraie pour tout entier positif i impair !

14. Notons que si i est pair, alors $t_i = 0$. En outre, pour savoir à partir d'un S_i, si i est pair, il suffit de regarder le cardinal de S_i : en effet, comme on part d'une partie vide et que l'on commute un unique interrupteur à chaque étape, le cardinal de S_i et i ont même parité.

Par ailleurs, même si la question précédente fait référence au minimum, **il ne faut pas oublier que les parties sont triées**, donc que le minimum est simplement la tête de la liste (qui ne peut être vide dans le cas qui nous intéresse puisque de longueur impaire).

Par conséquent, cela donne :

```
let succ_gray p =
  let ti = if List.length p mod 2 = 0 then 0 else List.hd p + 1
  in delta p [ti]
```

5 Système défaillant

15. Pour passer d'une partie p_1 à une partie p_2, on peut commencer par lever les interrupteurs de p_1 qui ne sont pas dans p_2, puis baisser ceux de p_2 qui ne sont pas dans p_1. À aucun moment on aura plus d'interrupteurs baissés que dans les configurations initiales et finales, qui sont valides.

On peut commencer par écrire une différence non symétrique $(P_1 \setminus P_2)$:

```
let rec diff p1 p2 = match p1, p2 with
  | _, [] -> p1
  | [], _ -> []
  | h1::t1, h2::t2 when h1<h2 -> h1::diff t1 p2
  | h1::t1, h2::t2 when h1=h2 -> diff t1 t2
  | h1::t1, h2::t2 -> diff p1 t2
```

Et donc

```
let test p1 p2 =
  diff p1 p2 @ diff p2 p1
```

16. Appliquons simplement la formule fournie (la complexité peut sembler importante du fait des concaténations et retournements, mais les tailles des listes croissent suffisamment vite pour que ce ne soit en pratique pas un réel problème si l'on y regarde de plus près).

```
let rec t n k =
  if n=1 then [0] else
  if k=1 then t (n-1) 1 @ [n-2; n-1] else
  t (n-1) k @ (n-1) :: (List.rev (t (n-1) (k-1)))
```

17. On a naturellement, d'après la définition de $\mathrm{T}(n,k)$:

$$\begin{cases} l_{1,k} = 1 \\ l_{n+1,1} = l_{n,1} + 2 \\ l_{n+1,k+1} = l_{n,k+1} + l_{n,k} + 1 \end{cases}$$

Notons que les deux premières relations donnent aisément $l_{n,1} = 2n-1$, et que la première et la troisième conduisent naturellement à $l_{2,k} = 3$ pour tout $k > 0$, et $l_{n,k} = 2^n - 1$ pour $n > 2$ et pour tout $k \geqslant n-1$. En particulier, $l_{n,n-1} = 2^n - 1$. Nous aurons besoin de cela plus loin.

Le nombre de configurations avec n interrupteurs où au plus k sont baissés est, si $k \geqslant 1$

$$s_{n,k} = \sum_{k'=0}^{\min(n,k)} \mathrm{C}_n^{k'}$$

En posant $\mathrm{C}_n^k = 0$ si $k > n$ ou $k < 0$, on a $s_{n,k} = \sum_{k'=0}^{k} \mathrm{C}_n^{k'}$.

On a donc

- $s_{1,k} = 2$;
- $s_{n,1} = n+1$ donc $s_{n+1,1} = s_{n,1} + 1$;
- $s_{n+1,k+1} = \sum_{k'=0}^{k+1} \mathrm{C}_{n+1}^{k'} = \sum_{k'=0}^{k+1} \mathrm{C}_n^{k'} + \sum_{k'=0}^{k} \mathrm{C}_n^{k'} = s_{n,k+1} + s_{n,k}$.

Puisque l'on cherche une relation entre $l_{n,k}$ et $s_{n,k}$, considérons $a_{n,k} = l_{n,k} - s_{n,k}$. On a, d'après les relations précédentes,

$$a_{n+1,k+1} = l_{n+1,k+1} - s_{n+1,k+1} = l_{n,k+1} + l_{n,k} + 1 - s_{n,k+1} - s_{n,k} = a_{n,k+1} + a_{n,k} + 1$$

Cette relation est proche de celle du triangle de Pascal, à un +1 près. Posons donc $b_{n,k} = a_{n,k} + 1 = l_{n,k} - s_{n,k} + 1$. On a

$$b_{n+1,k+1} = b_{n,k+1} + b_{n,k}$$

Par ailleurs, $b_{n+1,1} = (2n+1) - (1+n+1) + 1 = n$ et $b_{n+1,n} = 2^{n+1} - 1 - 2^{n+1} + 1 = 1$.

On a donc $b_{n+1,k} = \mathrm{C}_n^k$, soit $b_{n,k} = \mathrm{C}_{n-1}^k$, et donc

$$l_{n,k} = s_{n,k} + \mathrm{C}_{n-1}^k - 1$$

18. Montrons par récurrence que $\mathrm{P}_{n,k} = \mathcal{P}(\mathrm{I}_{n,k})$ où $\mathrm{I}_{n,k}$ est l'ensemble des parties de I_n de cardinal inférieur ou égal à k. L'idée est similaire à la question 10. On peut décomposer la séquence en :

$$\mathrm{S}_{k,0} \xrightarrow{\Delta\, t_0} \mathrm{S}_{k,1} \xrightarrow{\Delta\, t_1} \dots \xrightarrow{\Delta\, t_{l_{n,k}-1}} \mathrm{S}_{k,l_{n-1,k}} \xrightarrow{\{n-1\}} \mathrm{S}_{k,l_{n-1,k}+1} \xrightarrow{\Delta\, t_{l_{n-1,k}+1}} \dots \xrightarrow{\Delta\, t_{l_{n,k}-1}} \mathrm{S}_{k,l_{n,k}}$$

L'interrupteur $n-1$ n'est commuté qu'une seule fois dans cette séquence. En effet, avant le $\langle n-1, n-2\rangle$, on a une séquence $\mathrm{T}_{n-1,k}$ et ensuite une séquence $\tilde{\mathrm{T}}_{n-1,k-1}$ par construction.

On peut donc procéder par récurrence sur n et montrer que, pour chaque $n > 0$, pour tout $k > 0$, $\mathrm{P}_{n,k} = \mathcal{P}(\mathrm{I}_{n,k})$ et $\mathrm{S}_{n,l_{n,k}} = \{n-1\}$.

Pour $n = 1$, $l_{1,k} = 1$, on a $\mathrm{P}_{1,k} = \{\emptyset, \{0\}\}$, ce qui convient (on visite tous les états possibles avec un seul interrupteur et au plus un interrupteur baissé, et on termine avec l'interrupteur $n-1 = 0$ baissé).

Supposons cela vrai pour n, et considérons deux cas !

- Si $k = 1$, alors $\mathrm{P}_{n+1,1} = \{\emptyset, \{0\}, \emptyset, \{1\}, \dots, \emptyset, \{n\}\}$, et cela convient.
- Si $k > 1$, les $\mathrm{S}_{n+1,k}$ décrivent d'abord $\mathcal{P}(\mathrm{P}_{n,k})$, l'ensemble des états où n est levé à au plus k interrupteurs baissés, et termine avec $n-1$ baissé. Ensuite, on commute n (ce que l'on peut faire puisque $n-1$ est le seul baissé et $k > 1$, et on se retrouve avec n et $n-1$ seuls baissés et tous les autres interrupteurs levés. Puis on exécute une séquence qui ne touche plus à n et visite toutes les possibilités où l'on baisse au plus $k-1$ des $n-1$ interrupteurs restants (ce qui avec n fait k interrupteurs baissés), suivant exactement la séquence $\mathrm{P}_{n,k-1}$ à l'envers (on part de son état final où seul $n-1$ est baissé) jusqu'à relever tous les interrupteurs (sauf n donc).

On a donc bien $\mathrm{P}_{n,k} = \mathcal{P}(\mathrm{I}_{n,k})$ pour tout k et pour tout n.

19. Il s'agit en fait d'utiliser la liste $\mathrm{T}(\mathrm{N},\mathrm{K})$ en ajoutant simplement un $\mathrm{N}-1$ au bout

```
let test_panne n k =
  t n k @ [n-1]
```

20. Il faut évidemment au moins visiter l'ensemble des $s_{\mathrm{N},\mathrm{K}}$ configurations et revenir au départ, ce qui impose au moins $s_{\mathrm{N},\mathrm{K}}$ commutations. Mais pourquoi en faut-il davantage ? Pour le comprendre, il faut s'intéresser aux nombres de configurations avec au plus K interrupteurs baissés parmi N.

Par exemple, pour N = 6 et K = 3, il y a 1 configuration où ils sont tous levés, 6 où il y en a un baissé, 15 où il y en a deux, et 20 où il y en a trois. Donc 42 configurations.

Seulement, on constate que l'on a 26 configurations avec un nombre pair d'interrupteurs baissés, et seulement 16 avec un nombre impair. Or, en basculant un interrupteur, on ne peut passer que d'un nombre pair à un nombre impair ou inversement. Pour visiter les 26 configurations avec un nombre impair d'interrupteurs baissés, il faut donc au moins 52 étapes, soit 10 étapes de plus !

Le nombre de commutations est donc égal au moins au double du maximum du cardinal des configurations à un nombre d'interrupteurs pairs baissés et du cardinal des configurations à un nombre d'interrupteurs impair baissés. Il faut donc ajouter la valeur absolue de

la différence entre les deux cardinaux. Soit au moins

$$s_{\mathrm{N,K}} + \left| \sum_{k=0}^{\mathrm{K}} (-1)^k \mathrm{C}_{\mathrm{N}}^{k} \right| = s_{\mathrm{N,K}} + \left| (-1)^{\mathrm{K}} \mathrm{C}_{\mathrm{N}-1}^{\mathrm{K}} \right| = s_{\mathrm{N,K}} + \mathrm{C}_{\mathrm{N}-1}^{\mathrm{K}} = l_{\mathrm{N,K}} + 1$$

Résultats