Lycée Saint-Louis — DS 1
MPSI — Année 2024-2025

Devoir d'informatique n°1

Introduction

Nous considérons un système commandé par un tableau de bord comportant n interrupteurs, chacun pouvant être baissé ou levé. On désire tester ce système en essayant mécaniquement chacune des 2^n configurations possibles pour l'ensemble des interrupteurs. Le coût de cette opération sera le nombre total de mouvements d'interrupteurs nécessaires. Nous supposerons que chaque fois qu'un interrupteur est commuté le système effectue un diagnostic automatiquement et instantanément. Les interrupteurs sont indexés de 0 à $n-1$.

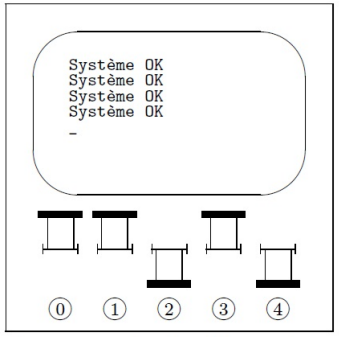

FIGURE 1. Pupitre de commande ; les interrupteurs 2 et 4 sont baissés.

Notations

Nous appelons *partie* un sous-ensemble fini de l'ensemble $\mathbb{N}$ des entiers naturels.
Un élément d'une partie est appelé *indice*. La *différence symétrique* de deux parties P et Q est définie par :

$$P \Delta Q = (P \setminus Q) \cup (Q \setminus P) = (P \cup Q) \setminus (P \cap Q)$$

On vérifie facilement que la différence symétrique est commutative et associative, ce que l'on ne demande pas de démontrer.

Pour tout n positif ou nul, nous notons $I_n = [\![0, n-1]\!]$ l'ensemble des entiers inférieurs strictement à n.
Une partie sera représentée par une liste chaînée d'indices distincts apparaissant dans l'ordre croissant des entiers.
On utilisera les types suivants :

```
type indice = int ;;
type partie = indice list ;;
```

Pour imprimer un entier i suivi d'un espace ou pour imprimer un saut de ligne, on pourra utiliser respectivement les instructions suivantes :

```
Printf.printf "%d " i;
print_newline ()
```

Partie 1 : Parties d'un ensemble

1. Écrire une fonction **card** de type `partie -> int` qui retourne le nombre d'éléments d'une partie.

   Les candidats devront résoudre cette question sans faire appel à la fonction de bibliothèque `List.length`.

2. Écrire une fonction **delta** de type `partie ->partie -> partie` qui réalise la différence symétrique de 2 parties. Le nombre d'opérations ne devra pas excéder $O(m+n)$, où m et n sont les cardinaux des arguments.

   Nous rappelons que dans toute liste chaînée de type `partie` les indices sont distincts et doivent apparaître dans l'ordre croissant des entiers.

3. Justifiez la terminaison et la correction de la fonction précédente.

4. Application au problème des interrupteurs : à chacune des configurations possibles, nous associons la partie formée des indices des interrupteurs baissés. Écrire un programme qui imprime la liste des indices d'interrupteurs à commuter pour passer d'une configuration à une autre.

   `val test : partie -> partie -> unit`

5. Donnez une estimation de la complexité $C(n)$ lorsqu'elle vérifie :
   (a) $C(n) = C(n-1) + O(1)$
   (b) $C(n) = C(n-1) + O(n)$
   (c) $C(n) = C(n/2) + O(1)$
   (d) $C(n) = C(n/2) + O(ln(n))$

   On justifiera très rapidement

Partie 2 : Énumération des parties par incrément

À toute partie P , nous associons l'entier $e(P) = \sum_{i \in P} 2^i$ (avec la convention $e(\emptyset) = 0$).

Nous définissons le successeur de P comme l'unique partie dont l'entier associé est $e(P) + 1$.

6. Déterminer $e(\{0, 1, 2, 5\})$ puis la partie correspondant à $e(\{0, 1, 2, 5\}) + 1$.
7. Écrire une fonction `chiffrage`

   ```
   val chiffrage : partie -> int
   ```
8. Quel est sa complexité ?
9. Si ce n'est pas le cas, la modifier pour que le nombre d'opérations n'excède pas $O(max)$ où max est le plus grand indice présent dans la partie donnée en argument.
10. Écrire une fonction `dechiffrage`

    ```
    val dechiffrage : int -> partie
    ```
11. Quel est sa complexité ?
12. Si ce n'est pas le cas, la modifier pour que le nombre d'opérations n'excède pas $O(\ln(e))$ où e est l'argument.
13. Écrire une fonction `succ` qui retourne le successeur d'une partie.

    ```
    val succ : partie -> partie
    ```
14. Quel est la complexité de `succ` dans le pire des cas ?
15. Écrire une fonction `succ2` qui retourne le successeur d'une partie. Le nombre d'opérations ne devra pas excéder $O(\ell)$ où ℓ est le plus petit indice absent dans la partie donnée en argument.

    ```
    val succ2 : partie -> partie
    ```
16. En application de ce mode d'énumération des parties, nous voulons réaliser le test de toutes les configurations d'interrupteurs. Au début et à la fin du test tous les interrupteurs seront levés.

    (a) Écrire la fonction `test_incr` qui imprime la liste des indices des interrupteurs à commuter pour réaliser la totalité du test pour interrupteurs et qui examine les configurations dans l'ordre défini par le successeur. L'argument de cette fonction sera l'entier n.

    ```
    val test_incr : int-> unit
    ```

    (b) Exprimer, en fonction de n, le nombre total d'interrupteurs à commuter pour réaliser le test de cette manière.

Partie 3 : Énumération des parties par un code de Gray

Nous notons $\langle u_0, \ldots, u_{k-1} \rangle$ une suite finie de k entiers. La concaténation de deux suites finies de longueur k et k' respectivement est une suite finie de longueur $k + k'$ définie par :

$$\langle u_0, \ldots, u_{k-1} \rangle \odot \langle u'_0, \ldots, u'_{k-1} \rangle = \langle u_0, \ldots, u_{k-1}, u'_0, \ldots, u'_{k-1} \rangle$$

La suite vide, notée $\langle\ \rangle$, est la suite de longueur 0. Une suite finie U est *préfixe* d'une autre suite finie V s'il existe une suite finie W telle que $V = U \odot W$ (autrement dit U est le début de V).

Pour tout entier positif ou nul n, nous considérons la suite finie $T(n)$ de longueur $2^n - 1$ définie par :

$$T(0) = \langle\ \rangle \text{ et } T(n+1) = T(n) \odot \langle n \rangle \odot T(n)$$

Nous avons par exemple $T(1) = \langle 0 \rangle$, $T(2) = \langle 0, 1, 0 \rangle$, et $T(3) = \langle 0, 1, 0, 2, 0, 1, 0 \rangle$.
Pour tout entier i positif ou nul nous notons t_i le $(i+1)$-ème élément de $T(n)$, s'il existe. Puisque $T(n)$ est préfixe de $T(n+1)$, la suite $(t_i)_{i \geq 0}$ est définie sans ambiguïté. Enfin, nous posons $S_0 = \emptyset$ et pour tout entier i positif ou nul nous définissons l'ensemble $S_{i+1} = S_i \Delta \{t_i\}$.

17. (a) Donner la valeur de $T(4)$.
    (b) Donner la valeur de S_i pour tout i inférieur ou égal à 15.
18. Nous voulons montrer que les S_i peuvent être utilisés pour énumérer les parties de I_n, et ainsi résoudre notre problème d'interrupteurs.
    (a) Donner la valeur de S_{2^n-1} pour tout $n \geq 0$.
    (b) Montrer que pour tout $n > 0$ et tout $i < 2^n$, on a : $S_{2^n+i} = S_i \Delta \{n-1, n\}$.
    (c) En déduire que pour tout $n \geq 0$, l'ensemble $\mathcal{P}_n = \{S_0, S_1, \ldots, S_{2^n-1}\}$ est l'ensemble des parties de I_n.
19. Application au problème des interrupteurs. Comme dans la deuxième partie, nous imposons que les interrupteurs soient levés au début et à la fin du test.
    (a) Écrire un programme s'inspirant des résultats de cette partie, qui imprime une liste d'indices d'interrupteurs à commuter pour réaliser la totalité du test. L'argument de cette fonction sera le nombre d'interrupteurs n.

    ```
    val test_gray : int -> unit
    ```

    (b) Quel est le coût du test avec cette méthode (c'est-à-dire le nombre total d'interrupteurs à commuter) ? Peut-on réaliser le test à un coût moindre ?
20. (a) Donner une expression de t_i en fonction de pour impair.
    (b) Écrire la fonction `gray` qui prend en argument une partie et retourne celle qui la suit immédiatement dans l'ordre défini par la suite $(S_i)_{i \geq 0}$.

    ```
    val gray : partie-> partie
    ```