DEVOIR SURVEILLÉ III

Informatique

Durée : 2 heures

Consignes :

- La candidate attachera la plus grande importance à la **clarté**, à la **précision** et à la **concision** de la rédaction. *2 points de la note finale sur 20 sont reservés au soin de la copie.*[1]
- Si une candidate est amenée à repérer ce qui peut lui sembler être une erreur d'énoncé, elle le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'elle a été amenée à prendre.
- **L'usage du cours, de notes, d'une calculatrice ou de tout autre appareil électronique est strictement interdit.**
- L'usage d'éléments du langage C non-encore vus en cours/TP/TD est interdit. Cela concerne notamment les boucles `for`, l'opérateur `++`, ainsi que `calloc` et `realloc`. *En cas de doute, demandez-moi.* L'utilisation de `else if` est autorisé.
- L'usage d'éléments du langage OCaml non encore vus en cours/TP/TD est interdit. Cela concerne notamment tous les aspects impératifs du langage (dont les `;`), ainsi que les paires/triplets/n-uplets. *En cas de doute, demandez-moi.*

Pour une bonne présentation de la copie, il est notamment demandé à la candidate de :

- Mettre en valeurs les résultats finaux.
- Tirer un trait horizontal entre chaque question afin que le début et la fin d'une question soient facilement identifiables.
- Indenter[2] ses codes.
- Si elle écrit un programme non-trivial, le préfixer d'un résumé de son fonctionnement en deux ou trois lignes. *Je m'autorise à mettre 0 à un code compliqué non expliqué, quand bien même il serait correct.*
- Prendre l'initiative de couper le code demandé en plusieurs fonctions si cela aide à la lecture.
- Numéroter ses copies en bas à droite sous le format numéro_de_la_page / nombre_total_de_pages.
- Ne pas utiliser de correcteur blanc.[3]

Dans tout le devoir, on supposera que les librairies C suivantes sont déjà incluses : `stdio.h` , `stdlib.h` , `stdbool.h` , `stdint.h` , `limits.h` . Quelques erreurs de syntaxe ponctuelles pourront être tolérées. Lorsque l'on programme en OCaml, on programme pour le compilateur `ocamlopt` et non pour l'interpréteur `utop` : en particulier, les `;;` sont interdits.

Les exercices sont indépendants.

Ne retournez pas la page avant d'y être invitées.

1. Ce n'est pas une lubie personnelle, cela se fait aux concours. Par exemple, CCINP maths réserve 1 point sur 20 pour le soin.
2. Le terme « indentation » désigne le décalage horizontal.
3. Car cela ne passe pas aux scanners des concours, et est en conséquence interdit aux concours.

Compléments de maths

Une suite est dite *arithmético-géométrique* s'il existe a et b tels que pour tous $n > 0$, $u_{n+1} = au_n + b$. Pour calculer la valeur des termes d'une telle suite :

- On commencer par calculer le point fixe de l'équation de récurrence, c'est à dire par résoudre l'équation suivante (d'inconnue l) : $l = al + b$.
- On pose la suite (appelée suite auxiliaire) $v_n = u_n - l$. On prouve qu'elle est géométrique, et on calcule la valeur de ses termes.
- On en déduit la valeur des termes de $u_n = v_n + l$.

I - Proche du cours

Exercice 1 – *Best of*

On considère la fonction ci-dessous :

somme-entiers.c

```c
/** Renvoie la somme des entiers de 1 à n (inclus).
  * Pas d'effets secondaires.
  */
int somme_entiers(int n) {
  int somme = 0;
  int i = 1;

  while (i <= n) {
    somme = somme + i;
    i = i +1;
  }

  return somme;
}
```

Cette fonction prend un entier supposé positif `n` en argument, et renvoie $\sum_{k=0}^{n} k$ sans effets secondaires.

1. Prouver la correction partielle de cette fonction (on ignore les éventuels débordements).
2. On rappelle que le type `int` est un type d'entiers signés. On le suppose de taille 4 octets (32 bits). Donner une valeur approximative du premier entier `n` positif pour lequel la fonction ne renvoie pas la bonne réponse. *On pourra passer rapidement sur les détails des calculs.*

Traitez l'exercice 2 ou l'exercice 3, mais pas les deux.

Exercice 2 – *Tours de Hanoï*

On rappelle le problème des tours de Hanoï. On dispose de trois *emplacements* (aussi appelés *tiges*) sur lesquels peuvent être empilés des disques. Les disques sont de diamètres deux à deux distincts. À aucun moment un disque plus grand ne peut être placé au-dessus d'un disque plus petit.
On dispose d'une seule opération pour déplacer les disques : on peut prendre le disque au sommet d'un emplacement, et le poser au sommet d'un autre emplacement. On nomme cette opération `DÉPLACE(depart, arrivee)` (on déplace le disque du sommet de l'emplacement `depart` à l'emplacement `arrivee`).

1. Écrire **en pseudo-code** une fonction `HANOI(depart, arrivee, k)` qui déplace les `k` premiers disques de l'emplacement `depart` vers l'emplacement `arrivee`.
   On impose comme précondition que `depart` ≠ `arrivee`, que l'emplacement de départ contienne bel et bien au moins `k` disques, et que les disques des emplacements autres que `depart` sont tous plus grand que les `k` premiers disques de `depart`.
2. On veut évaluer sa complexité en nombre d'appels à `DÉPLACE`. On note $D(k)$ le nombre d'appels à `DÉPLACE` effectué par `HANOI(depart, arrivee, k)`.
   Écrire $D(n)$ comme une suite arithmético-géométrique, et la résoudre. *(Vous pouvez aller vite sur les détails des calculs : je ne suis pas prof de maths.)*

On admet que $D(n)$ est indépendant du départ et de l'arrivée, çad qu'il dépend uniquement de n.

3. On suppose désormais que l'une des tiges est penchée. Cela permet de déplacer autant de disques que l'on veut depuis cette tige vers une autre en 1 seul déplacement (tous les disques ainsi déplacés doivent aller sur la même tige). On nomme `SUPER-DÉPLACE(depart, arrivee, i)` le fait de déplacer `i` disques d'un coup de `depart` à `arrivée`.
   Proposez, **en pseudo-code**, une fonction `PISE(depart, arrivee, k)` qui améliore `HANOI` en exploitant cette tige penchée. Sa spécification est la même que `HANOI` (mis à part qu'il existe maintenant une tige penchée).
4. Exprimez la ou les équations qui définissent la complexité pire des cas de cette fonction (complexité comptée en nombre d'appels à `DÉPLACE`/`SUPER-DÉPLACE`). Il n'est pas attendu que vous résolviez le système.

Exercice 3 – *Compteur binaire*

On considère un compteur binaire, c'est à dire un entier écrit en binaire non-signé. On note L son nombre de bits ; L est fixé et ne variera pas. On considère une opération `INCR` qui augmente de 1 l'entier signé. En cas de débordement, on ignore les retenus sortantes (comme fait `unsigned` en C).
On dit qu'on « flip » un bit quand on change sa valeur. On s'intéresse à la complexité de `INCR` en nombre de flips. Cela donne par exemple les coûts suivants :

0	0	0	0	0	0	0	0	$\xrightarrow{\texttt{INCR}}$	0	0	0	0	0	0	0	**1**

Coût : 1 flip (en rouge et gras)

1	0	1	0	1	1	1	1	$\xrightarrow{\texttt{INCR}}$	1	0	1	**1**	**0**	**0**	**0**	**0**

Coût : 5 flips (en rouge et gras)

1. **a.** Quelle est la complexité pire des cas d'un appel à `INCR` ?
   b. En déduire un majorant du coût de K appels successifs à `INCR`. Expliquer en quoi cette majoration est trop grossière.

On considère donc une suite de K appels à `INCR` (avec K grand). L'état initial du compteur est le compteur à 0 (tous les bits sont nuls).

2. Expliquer pourquoi à chaque `INCR` il y a au plus un bit flippé de 0 à 1 .
3. Par la méthode du comptable (et pas une autre !), prouvez à l'aide de la question précédente qu'au sein d'une suite de K appels à `INCR`, chaque appel a un coût amorti constant.

II - Vrac à exercices

Pas de gros problème dans ce DS-ci, mais 2 petits exercices de difficulté à peu près croissante.

Exercice 4 – *Écart maximal dans une liste*

Les questions de programmation sont à traiter en OCaml.

1. Écrire une fonction `ecart : int -> int -> int` qui calcule la valeur absolue de la différence entre deux entiers.
   Par exemple, `ecart 5 3` renvoie `2` et `ecart 3 5` de même.

On voudrait maintenant trouver l'écart maximal entre deux éléments d'une liste. Pour cela, on va calculer le maximum et le minimum de la liste, puis en calculer l'écart.

2. Prouvez mathématiquement que cette méthode calcule bien l'écart maximal. Autrement dit, prouver que pour tout ensemble E fini non-vide d'entiers :

$$|\min(E) - \max(E)| = \max\{|x - y| \; t.q. \; x \in E \; et \; y \in E\}$$

En OCaml sont pré-codées les fonctions `min : 'a -> 'a -> 'a` et `max : 'a -> 'a -> 'a` qui renvoient respectivement le minimum et le maximum de leurs deux arguments. Ainsi, `min 1 0` renvoie `0` et `min 0 1` de même.

3. À l'aide de la signature de ces fonctions, expliquez pourquoi le code OCaml ci-dessous ne résout pas le problème de l'écart maximal dans une liste :

max-ecart.ml
```
4  (* il suffit d'appliquer min, trop facile ! *)
5  let trop_facile = fun lst ->
6    match lst with
7    | [] -> failwith "min liste vide"
8    | t::q -> min lst (* si la liste n'est pas vide, easy *)
9
10 let _ =
11   print_int (trop_facile [5;3;2])
```

4. Dans le code à trou ci-dessous, le motif `t::[]` sert à identifier les listes ayant un seul élément. Si on rentre dans ce motif, on ne rentre pas dans les suivants (car les motifs sont testés de haut en bas et on utilise le premier qui correspond).
   Proposer une expression OCaml pour compléter la ligne 18 de sorte à ce que cette fonction renvoie le minimum d'une liste non-vide :

max-ecart.ml
```
14 let rec min_list = fun lst ->
15   match lst with
16   | []      -> failwith "min liste vide"
17   | t :: [] -> t
18   | t :: q  ->
```

On construit de même une fonction `max_list` .

5. En déduire une fonction `ecart_maximal_list : int list -> int` qui renvoie l'écart maximal entre deux éléments d'une liste d'entiers prise en argument. Votre complexité doit être linéaire en la longueur de la liste (il n'est pas nécessaire de le prouver).
6. Refaites la question précédente mais en utilisant un seul parcours de la liste. Ainsi, vous ne pouvez *pas* appeler `min_list` *et* `max_list` car chacune d'entre elles parcourt la liste une fois, donc au total les deux appels parcourent la liste deux fois.

Exercice 5 – ~~*quick*~~*slowselect*

Les questions de programmation sont à traiter en OCaml. On pourra utiliser les deux fonctions ci-dessous. Elles sont proviennent de la librairie `List`, d'où le fait que leur nom débute par `List.` .

- `List.length : 'a list -> int` renvoie la longueur d'une liste.
- `List.filter : ('a -> bool) -> 'a list -> 'a list` est telle que `List.filter f lst` est la liste des éléments de `lst` sur lesquels `f` s'évalue à `true` . Elle ne réordonne pas ces éléments.
  Ainsi, `List.filter (fun x -> x < 5) [ 3; -20; 65; 1; 23; 0]` vaut `[ 3; -20; 1; 0]` .
  Similairement, `List.filter (fun x -> x < 5) []` vaut `[]` .

1. **a.** Proposez une implémentation de `length` .
   b. Proposez une implémentation de `filter` .
   Indication : si `lst` *est de la forme* `t::q`*, la filtration du tout est la filtration de la queue à laquelle s'ajoute ou non l'élément* `t`

Dans la suite de cet exercice, pour simplifier, on suppose que les éléments des listes sont deux-à-deux distincts. On s'intéresse au problème suivant :

- Entrée : `lst` une liste (non-vide), et $i \in [\![0;$ `List.length` *lst*$[\![$.
- Tâche : trouver le $i$ème plus petit élément de la liste (le plus petit est le 0ème, ..., et le plus grand le (`List.length lst -1`)-ème).

Exemple. Le 3ème plus petit élément de `[0; 22; 15; 20; -2; 35; 7; 3; 28]` est `7` . (Notez qu'il s'agit en français du « quatrième » plus petit, puisque l'on compte à partir du 0ème.)

On propose le pseudo-code suivant :

Fonction slowselect

Entrées : i et *lst* (non-vide) comme décrits ci-dessus
Sorties : Le $i$ème plus petit élément de lst

```
1  pivot ←élément de tête de lst
2  smaller ←éléments de lst strictement inférieurs à pivot
3  card_smaller ←longueur de la liste smaller
4  bigger ←éléments de lst strictement supérieurs à pivot
5  si i = card_smaller alors
6  |  renvoyer pivot
7  sinon si i < card_smaller alors
8  |  renvoyer SLOWSELECT(i, smaller)
9  sinon
10 |  renvoyer SLOWSELECT(i − 1 − card_smaller, bigger)
```

2. Déroulez cet algorithme sur l'entrée `i` = 3 et `lst` = `[0; 22; 15; 20; -2; 35; 7; 3; 28]`.
   La représentation de la pile mémoire n'est *pas attendue*. Ce qui est attendu, c'est de montrer clairement :
   - Quels appels récursifs sont faits.
   - Au sein de chaque appel récursif, quelles sont les valeurs d'entrée de i et de `lst`.
   - Au sein de chaque appel récursif, quelles sont les valeurs de `pivot`, `smaller` et `bigger`. Il n'est pas nécessaire d'indiquer la valeur de `card_smaller`.
   - Au sein de chaque appel récursif, quelle branche du `Si-SinonSi-Sinon` est utilisée. On peut nommer les trois branches A, B et C (de haut en bas) et simplement dire "branche A", "branche B", ou "branche C".
3. Prouvez la terminaison de cette fonction. On pourra utiliser la longueur de `lst` comme variant d'appels récursifs.
4. Prouvez la correction partielle de cette fonction : il s'agit de prouver par récurrence (forte) sur la longueur de `lst` que « si i vérifie les pré-conditions, SLOWSELECT(`lst`, i) est le $i$ème plus petit élément de `lst` ».
   On n'oubliera pas de justifier que les appels récursifs sont faits sur des listes non-vides.
5. Codez cette fonction en OCaml. Les fonctions `List.length` et `List.filter` seront utiles. Si jamais la liste donnée en entrée est vide, on pourra renvoyer `failwith "slowselect : liste vide"`
6. Calculez la complexité de cette fonction OCaml, en nombre de comparaisons.
   Vous commencerez par évaluer la complexité en nombre de comparaisons des appels à `List.filter` et `List.length` de votre code.
   Indication : vous voulez majorer la complexité d'un pire des cas. Analysez donc le cas où la longueur de `lst` *décroit le plus lentement possible.*

Remarque. L'algorithme présenté ici est une version simplifiée d'un algorithme nommé `quickselect`. Pour être plus rapide, `quickselect` choisit comme pivot la médiane de la liste : elle coupe la liste en deux moitiés égales et évite le pire des cas de complexité analysé ci-dessus. Toute la difficulté est alors de calculer cette médiane efficacement : cela peut se faire en temps linéaire, et on obtient un `quickselect` qui s'exécute en temps linéaire.

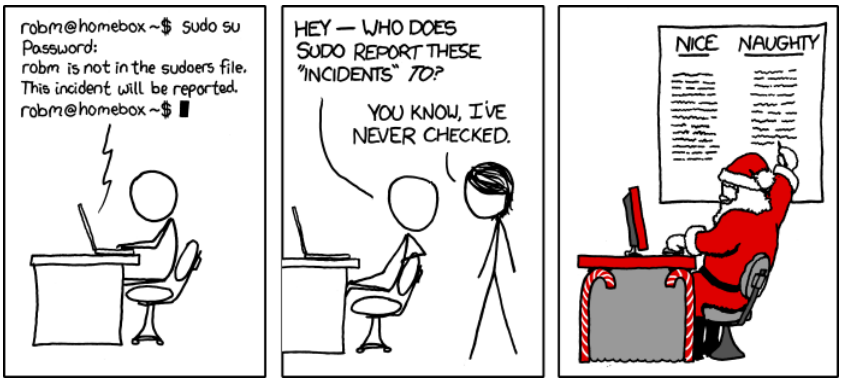

Figure III.1 – https://xkcd.com/838/