Correction du DS 2

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici...

Exercices

Exercice α1 :

Soit v l'expression dont on a déjà montré qu'elle décroissait strictement une fois sur deux et restait identique l'autre fois. On pose v' la valeur $2v+1$ si v a décru au tour de boucle précédent et $2v$ sinon. Alors v' est toujours un entier positif, et il décroît strictement à chaque tour de boucle, donc il s'agit d'un variant. Il est important de noter qu'un variant peut ne pas dépendre uniquement des variables du programme, mais annoncer « le nombre de tours de boucle restants » ne permet pas de disposer d'un variant car ceci peut ne pas être un entier... si c'est infini.

Exercice α2 :

D'après la formule du cours, la réponse est $\Theta(n)$. Il s'agit par exemple d'une tentative malheureuse de dichotomie qui procède à une récursion sur un tableau, en passant une moitié du tableau en argument de l'appel suivant, d'où un coût de recopiage. En pratique la réponse se voit tout de suite sur la formule : c_n est la somme d'un $\Theta(n)$ et d'une valeur positive.

Exercice α3 :

Entre autres :

```
chmod o+r copie
```

Exercice β1 :

```
let rec fold_left f x l =
  match l with
  | [] -> x
  | a::q -> fold_left f (f x a) q;;
```

Exercice β2 :

Principe de l'algorithme : on utilise deux indices, un qui démarre à la fin du tableau et qui décroît, l'autre qui démarre au début du tableau et qui croît.

On commence par faire décroître celui de droite jusqu'à trouver `x`, puis on fait croître celui de gauche jusqu'à trouver `x`, et ainsi de suite en alternant jusqu'à ce que les indices se soient rejoints, forcément sur un indice où se trouve `x` ou sur la position de départ de celui de gauche si `x` n'est pas dans le tableau.

Le test supplémentaire quand on recule est pour éviter une erreur dans le cas d'un tableau ne contenant pas `x`, cette précaution n'étant pas nécessaire quand on avance car `x` aura alors déjà été trouvé.

On note que le tableau vide est bien géré par la même précaution.

```
let position_mediane t x =
  let debut = ref (-1) in
  let fin = ref (Array.length t) in
  let avancer = ref false in
  while !debut < !fin do
    if !avancer then
    begin
      incr debut;
      if t.(!debut) = x
        then avancer := false
    end
    else
    begin
      decr fin;
      if !fin > !debut && t.(!fin) = x
        then avancer := true
    end
  done; if !fin >= 0 then Some !fin else None;;
```

Exercice β3 :

On peut lancer deux dichotomies, l'une entre le début et la position de la valeur qu'on a trouvée en premier (par une dichotomie simple), l'autre entre cette même position et la fin, le tout après la première dichotomie dans la même fonction. Pour se faciliter la vie, il est tout à fait envisageable de séparer le travail en deux fonctions, sans préjudice sur la complexité. Ceci étant, comme il risque fort d'y avoir des redondances de type « programmation par copier-coller », une seule fonction sera écrite avec comme paramètre un prédicat dont on cherche le moment où il est vérifié (la supériorité à `x` au sens large puis la supériorité au sens strict).

```
let premier_indice t pred =
  let deb = ref 0 in
  let fin = ref (Array.length t) in
  while !deb < !fin do
    let milieu = (!deb + !fin) / 2 in
    if pred t.(milieu) then fin := milieu
    else deb := milieu + 1
  done;
  !deb;;

let intervalle t x =
  let deb = premier_indice t (fun elt -> elt >= x) in
  let fin = premier_indice t (fun elt -> elt > x) in
  if deb = fin then (-1, -1)
  else (deb, fin-1);;
```

Exercice γ1 :

```
void echange(int n, int t1[], int t2[])
{
  for (int i = 0 ; i < n ; i += 1)
  {
    int buff = t1[i];
    t1[i] = t2[i];
    t2[i] = buff;
  }
}
```

Exercice γ2 :

```
int amplitude_max(int n, int t[])
{
  int rep = t[1] - t[0];
  int repabs = rep > 0 ? rep : -rep;
  for (int i = 2 ; i < n ; i += 1)
  {
    int ecart = t[i] - t[i-1];
    if (ecart > repabs || ecart < -repabs)
    {
      rep = ecart;
      repabs = rep > 0 ? rep : -rep;
    }
  }
  return rep;
}
```

Exercice γ3 :

On remercie la fonction `strcmp` de `string.h` de faire l'essentiel du travail.

```
int max_lexico(int n, char* tab[])
{
  int rep = 0;
  for (int i = 0 ; i < n ; i += 1)
  {
    if (strcmp(tab[i], tab[rep]) > 0) rep = i;
  }
  return rep;
}
```

Problème

Question P1 :

En prenant pour a, b et n des nombres premiers, on évite le risque que des valeurs de u_k deviennent à terme impossibles.

Bien entendu, les nombres premiers doivent être distincts.

Question P2 :

En prenant pour n une puissance de deux, le calcul du reste correspond à l'extraction des derniers bits, ce qui est plus facile et donc plus rapide

Il faut alors envisager un post-traitement de u_k, car la parité est par exemple bien trop régulière.

Question P3 :

Une seule division euclidienne permet d'économiser du temps par rapport à une par étape, mais les nombres manipulés sont plus grands, donc le gain est moins important que l'on pourrait croire. Et puisque les nombres sont plus grands, ils risquent fort de déborder de l'intervalle des entiers représentables après peu d'étapes (vu les valeurs classiques pour a, b et n, c'est éventuellement dès la deuxième étape que cela se produit pour des entiers sur 32 bits), d'où le choix de faire les divisions euclidiennes à chaque étape. La justification de l'égalité se fait en maths.

Question P4 :

```
let gcl u0 a b n t =
  let rep = Array.make t u0 in
  let uk = ref u0 in
  for i = 0 to t-1 do
    uk := (a * !uk + b) mod n;
    rep.(i) <- !uk
  done; rep;;
```

Question P5 :

Quand on élève au carré un nombre à dix chiffres (dont les premiers peuvent certes être des zéros), on obtient un nombre ayant jusqu'à vingt chiffres (y compris sans zéro initial, cette fois). Il faut plus de 64 bits pour écrire les plus grands tels nombres, ce qui est incompatible avec les types `int` en OCaml et en C. (En pratique, quand le calcul est exact modulo 2^n avec $n = 63$ en OCaml et $n = 64$ en C pour les entiers longs non signés, les débordements ne sont pas problématiques, mais on évitera tout de même d'avoir à se poser la question. Attention, pour les entiers signés, il n'y a pas de garantie sur le comportement en cas de débordement en C.)

Question P6 :

En utilisant huit chiffres, l'élevation au carré donne un nombre inférieur à 10^{16}, donc tenant sur une cinquantaine de bits. Les types `unsigned long int` et `long int` conviennent en C.

```
long int* vng(long int u0, int t) // long non nécessaire
{
  long int* rep = malloc(t * sizeof(long int));
  long int uk = u0;
  for (int i = 0 ; i < t ; i += 1)
  {
    uk = ((uk * uk) / 10000) % 100000000; // ici long nécessaire pour uk * uk
    rep[i] = uk; // si int dans le tableau, caster par principe
  }
  return rep;
}
```

Question P7 :

```
int* test1(int l, int tab[], int n)
{
  int* rep = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1) rep[i] = 0;
  for (int i = 0 ; i < l ; i += 1) rep[tab[i]] += 1;
  return rep;
}
```

Question P8 :

Ici on renvoie bêtement la matrice. On peut aussi lancer une récursion pour récupérer une liste comme suggéré.

```
let test2 tab n =
  let rep = Array.make_matrix n n 0 in
  for i = 0 to Array.length tab - 2 do
    rep.(tab.(i)).(tab.(i+1)) <- rep.(tab.(i)).(tab.(i+1)) + 1
  done;
  rep;;
```