Correction et terminaison

I - Proche du cours

Exercice 1

1. Écrire un algorithme qui prend en entier un tableau d' entiers et sa longueur ℓ et renvoie le minimum du tableau.
2. Prouver qu'il termine.
3. Prouver qu'il est totalement correct.

Exercice 2

On suppose qu'on dispose de deux fonctions `max` et `min` qui renvoient respectivement le maximum et le minimum de deux entiers.
On considère la fonction `snd_max` suivante :

Entrées : Un entier $0 < l$ et un tableau T contenant l entiers.

Sortie : Le deuxième plus grand élément de T.

pdp.c

```c
19 /** Spécification : mystère mystère */
20 int snd_max(int l, int T[]) {
21   int maxi = max(T[0], T[1]);
22   int snd_maxi = min(T[0], T[1]);
23   int i = 2;
24   while (i < l) {
25     if (T[i] > maxi) {
26       snd_maxi = maxi;
27       maxi = T[i];
28     }
29     else if (T[i] > snd_maxi) {
30       snd_maxi = T[i];
31     }
32     i = i+1;
33   }
34   return snd_maxi;
35 }
```

1. (facultatif) Pour comprendre l'algorithme, vous pouvez le faire tourner à la main sur un exemple de votre choix, par exemple $[|1;3;7;5;8;10;4;2;15|]$.
2. Montrer que cet algorithme termine.
3. Montrer que cet algorithme est partiellement correct. Conclure.
4. Est-ce toujours le cas si on inverse les lignes 26 et 27 ?

II - Plus avancé

Exercice 3

Recherche dichotomique :
On considère l'algorithme suivant, qui recherche par dichotomie[i] un entier dans un tableau :

Entrées : Un entier n, un tableau T contenant n valeurs triées par ordre croissant et un entier x.

Sorties : Un indice $i \in [\![0; n-1]\!]$ tel que `T[i]`$= x$ si un tel indice existe, -1 sinon.

pdp.c

```c
/** Recherche dichotomique de x dans T.
 * Entrées : T un tableau de len entiers, trié croissant
 *           x un entier
 * Sortie : si x est dans T, un indice où il se trouve;
 *          sinon -1
 */
int recherche(int len, int T[], int x) {
  int deb = 0, fin = len-1;
  int milieu;
  while (fin - deb >= 0) {
    milieu = (deb + fin)/2;
    if (T[milieu] == x) {
      return milieu;
    }
    else if (T[milieu] < x) {
      deb = milieu+1;
    }
    else {
      fin = milieu - 1;
    }
  }
  return -1;
}
```

1. Faites tourner cet algorithme sur l'exemple suivant : T = [2 ;5 ;10 ;16 ;17 ;17 ;23] et x = 10.
2. Montrer que cet algorithme termine.
3. Montrer que si l'algorithme renvoie un indice $i \neq -1$, alors ce résultat est correct.
4. On note T[a : b[le tableau entre les cases a et $b-1$: T[a : b[= {T[a] ; … ; T[b-1]} (attention, on exclut l'indice b). Montrer qu'on a l'invariant suivant : « x n'est ni dans $T[0 : \texttt{deb}[$ ni dans $T[\texttt{fin}+1 : n[$ ».
5. En déduire la correction de l'algorithme.
6. L'algorithme reste-t-il correct (totalement ? partiellement ?) si on remplace la ligne `deb = milieu + 1` par `deb = milieu` ?
7. On manipule maintenant de gros entiers et on suppose que la longueur du tableau est proche du plus grand entier que l'on peut encoder dans notre type (par exemple, $n = 2^{32} - 2$ et l'on encode n dans un entier non signé sur 32 bits).
   a. Peut-on avoir des dépassements de capacité ? À quelle(s) ligne(s) ?
   b. Proposer une modification de l'algorithme pour éviter les éventuels dépassements. On ne touchera pas à l'encodage des entiers (on garde des entiers sur 32 bits).

i. La recherche dichotomique sur un ensemble trié consiste à réduire de moitié l'espace de recherche à chaque itération en comparant la valeur recherchée au milieu.

Exercice 4

On considère la fonction `est_permutation` suivante :

Entrées : Un entier $len \leq 1000$ et un tableau T contenant len entiers.

Sorties : Un booléen qui vaut `true` si T est une permutation de $[\![0; n-1]\!]$ et `false` sinon. (T est une permutation $[\![0; n-1]\!]$ si et seulement si il contient exactement une fois chaque valeur entre 0 et $n-1$

inclus.)

pdp.c

```c
84  /** Teste si un tableau est une permutation.
85   * Entrées : T un tableau de len cases, len <= 1000.
86   * Sortie : true SSI T contient chaque entier de [0; len[
87   *                   une et une seule fois
88   */
89  bool est_permutation(int len, int T[]) {
90    int compte[1000] = {}; // 1000 cases à 0
91    int indice = 0;
92    while (indice < len) {
93      compte[T[indice]] += 1;
94      indice = indice +1;
95    }
96    indice = 0;
97    while (indice < len) {
98      if (compte[indice] != 1) {
99        return false;
100     }
101     indice = indice +1;
102   }
103   return true;
104 }
```

1. (facultatif) Pour comprendre l'algorithme, vous pouvez le faire tourner à la main sur un exemple de votre choix, par exemple $[|1; 3; 7; 5; 8; 9; 4; 2; 6; 0|]$.
2. Prouver la terminaison.
3. Que contient le tableau `compte` en sortie de la première boucle (ligne 96) ? Prouvez le (avec un invariant).
4. Prouver que l'algorithme est totalement correct.

Exercice 5

Exponentiation rapide (version itérative) :
On considère la fonction `expo` qui prend en argument un entier a quelconque et un entier $n \geq 0$ et renvoie l'entier a^n.
L'idée de l'exponentiation rapide est la suivante : on décompose n en binaire.
L'exponentiation rapide fonctionne en décomposant n en binaire. Faisons un petit exemple : si $n = 6^2\overline{110}$. Alors $a^n = a^{{}^2\overline{110}}$. On utilise alors ${}^2\overline{110} = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Ainsi :

$$a^6 = a^{2^2+2^1} = a^{2^2} \times a^{2^1}$$

L'algorithme itératif d'exponentiation rapide utilise ce principe. On mémorise :

- `acc` qui stocke le a^{2^i} en cours.
- `res` qui stocke le produit de a^{2^j} déjà fait.
- `puiss` : initialisé à n, cette variable permet d'accéder aux bits successifs de n en la divisant par 2 à chaque itération.

pdp.c

```c
/** Exponentation rapide itérative.
 * Entrées : a entier, n entier positif.
 * Sortie : a^n.
 */
int expo(int a, int n) {
  int puiss = n, res = 1, acc = a;
  while (puiss != 0) {
    if (puiss % 2 == 1) {
      res = res*acc;
    }
    acc = acc*acc;
    puiss = puiss/2;
  }
  return res;
}
```

1. Faites tourner l'algorithme à la main sur 2^{25}. Efforcez-vous de faire le lien avec les explications de l'énoncé.
2. Montrer que $\texttt{res} \times \texttt{acc}^{\texttt{puiss}} = a^n$ est un invariant de boucle.
3. En déduire la correction partielle de la fonction.
4. Montrer la correction totale de la fonction.
5. (bonus) Si on encode tous nos entiers dans des `uint8_t` et qu'on prend en compte les dépassements de capacité, sur quels couples d'entrées (a, n) cet algorithme reste-t-il correct ?
6. (Bonus, NSI Terminale) Est-ce le même algorithme que l'exponentiation rapide récursive ? Pourquoi ?