TP : 5 algorithmes importants

Question 1.

Écrire la fonction `int max_array(const int *a, size_t n)` qui renvoie le maximum d'un tableau d'entiers de taille n. Faire une étude rapide de sa complexité.

Remarque.
- — Les tailles et les indices des tableaux peuvent être données sous la forme d'entiers `int` ou bien utiliser le type plus adapté `size_t` (qui désigne des entiers non signés).

  Le spécifieur de format pour ce type est `%zu`.
- — En C, lorsqu'on passe un tableau en paramètre de fonction, il se décaye en pointeur. Donc ces trois écritures sont strictement équivalentes pour le compilateur :

```c
int f(int a[], size_t n);
int f(int *a, size_t n);
int f(const int *a, size_t n);

```

- — On emploie ici `const int*` au lieu de `int a[]` pour indiquer qu'on n'a pas l'intention de modifier le tableau (et que ça ne devrait pas arriver).

Solution. Hors de la boucle : 4 instructions/expressions en $O(1)$.

Le corps de la boucle (4 instructions/expressions en $O(1)$) est également en $O(1)$ et il y a n passages (jamais moins).

Donc la complexité temporelle au pire (qui est égale à la complexité au mieux) est un $O(n)$ ($O(4n+4)$ soit $O(n)$).

```c
#include <assert.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h> //pour memcmp
#include <math.h> //pour fabs

int max_array(const int *a, size_t n) {
    assert(n > 0);
    size_t i = 1;
    int m = a[0];
    while (i < n) {
        if (a[i] > m) m = a[i];
        i++;
    }
    return m;
 }

void text_max(void) {
    int t1[] = {3, 1, 7, -2, 7};
    assert(max_array(t1, 5) == 7);

    int t2[] = {-5, -10, -3};
    assert(max_array(t2, 3) == -3);
}

```

□

Question 2.

Écrire la fonction `double avg_array(const double *a, size_t n)` qui calcule la moyenne arithmétique d'un tableau non vide. Donner sa complexité temporelle.

Solution. Complexité : voir plus haut.

Code

```c
/* 2) avg : moyenne arithmétique d'un tableau non vide */
double avg_array(const int *a, size_t n) {
    assert(n > 0);
    size_t i = 0;
    long long s = 0;
    while (i < n) {
        s += a[i];
        i++;
    }
    return (double)s / (double)n;
}

void test_avg(void) {
    int t1[] = {2, 2, 2, 2};
    assert(fabs(avg_array(t1, 4) - 2.0) < 1e-12);
    // fabs : valeur absolue des flottants
    // < 1e-12 : suivant l'ordre dont on fait les calculs de flottants,
    // les résultats peuvent être légèrement différents
    int t2[] = {1, 2, 3, 4};
    assert(fabs(avg_array(t2, 4) - 2.5) < 1e-12);
}

```

□

Question 3.

Écrire la fonction `int search_array(const int *a, int n, int target)` qui renvoie la position de la cible (-1 si on ne la trouve pas). Donner sa complexité temporelle.

Solution. Complexité :

- Dans le pire des cas, on ne trouve pas la cible. Il faut donc parcourir tout le tableau et l'analyse faite pour `max_array` reste valable.
- Dans le meilleur des cas, n trouve immédiatement la cible. Il n'y a qu'un passage dans la boucle. Les instructions hors de la boucle sont en $O(1)$ et le corps de boucle aussi.
  Conclusion : complexité temporelle en $O(1)$

Code

```c
/* 3) search : renvoie l'indice ou on trouve la cible, -1 sinon */

int search_array(const int *a, size_t n, int target) {
    size_t i = 0;
    while (i < n) {
      if (a[i] == target) return (int) i;
        i++;
    }
    return -1;
}

void test_search(void) {
    int t1[] = {5, 8, 13, 8};
    assert(search_array(t1, 4, 13) == 2);

    int t2[] = {9, 9, 9};
    assert(search_array(t2, 3, 7) == -1);
}

```

□

Question 4.

Écrire la fonction `void rev_array(int *a, size_t n)` qui inverse l'ordre des éléments d'un tableau (*en place* donc sans utiliser de tableau auxiliaire). Donner sa complexité temporelle.

Solution. Complexité : les instructions hors boucle sont en $O(1)$. La totalité du corps de boucle aussi. On ne parcourt que la moitié du tableau (et c'est toujours ainsi car pas de meilleur cas) : il y a donc $n/2$ passages en $O(1)$*chacun*. La complexité est en $O(n)$ puisque $n/2$ et n sont proportionnels.

Code

```c
/* 4) rev : inversion en place du tableau */
void rev_array(int *a, size_t n) {
    size_t i = 0;
    while (i < n / 2) {
        int tmp = a[i];
        a[i] = a[n - 1 - i];
        a[n - 1 - i] = tmp;
        i++;
    }
}

void text_rev(void) {
    int t1[] = {1, 2, 3, 4, 5};
    int exp1[] = {5, 4, 3, 2, 1};
    rev_array(t1, 5);
    assert(memcmp(t1, exp1, sizeof t1) == 0);

    int t2[] = {10, 20, 30, 40};
    int exp2[] = {40, 30, 20, 10};
    rev_array(t2, 4);
    assert(memcmp(t2, exp2, sizeof t2) == 0);
 }

/*

On teste un tableau de longueur paire et un autre de longueur paire.
Toujours procéder ainsi si on ne parcourt que la 'moitié' d'un tableau

*/


/*
    memcmp(const void *s1, const void *s2, size_t n) compare les n
    premiers octets de deux zones mémoire.

    Retourne 0 si les zones sont identiques.

    Retourne <0 si la première zone est 'plus petite' que la
    seconde.

    Retourne >0 si elle est 'plus grande'.

     Dans nos tests, on
     utilise memcmp(...) == 0 pour vérifier que deux tableaux sont
    identiques.
*/

```

□

Question 5.

Écrire la fonction `bool is_sorted_array(const int *a, size_t n)` qui renvoie vrai si le tableau est trié par ordre croissant et faux sinon. Donner sa complexité temporelle.

Solution. Complexité au pire et au meilleur : comme pour `search_array`.

Code

```c
/* 5) is_sorted : true si trié croissant, 0 sinon */
bool is_sorted_array(const int *a, size_t n) {
    size_t i = 1;
    while (i < n) {
        if (a[i - 1] > a[i]) return false;
        i++;
    }
    return true;
}

void test_is_sorted(void) {
    int t1[] = {1, 2, 2, 4};
    assert(is_sorted_array(t1, 4));

    int t2[] = {3, 1, 2};
    assert(!is_sorted_array(t2, 3));
}


```

□

Solution. Code du `main` :

```c
/* Lance toutes les batteries de tests */
int main(void) {
    test_max();
    test_avg();
    test_search();
    test_rev();
    test_is_sorted();
    puts("Tous les tests passent");
    return 0;
}

```

□