EPITA S4 Promo 2019
Programmation - Épreuve machine

Marwan Burelle*

9 mai 2016

Instructions :

Vous devez lire l'intégralité du sujet, ainsi que l'ensemble de ces instructions. Tout ce qui est explicitement demandé dans ce sujet est obligatoire. La perte de points dûe au non respect des consignes explicites ne sera pas contestable.

Votre répertoire pendant l'épreuve est temporaire, dans ce répertoire vous trouverez un répertoire `subject` *et un répertoire* `submission`*. Dans le répertoire* `submission` *vous trouverez tous les fichiers pour faire votre exam (squelette des questions.)*

Vous devez effectuer des rendus réguliers pour être sûr de ne pas perdre votre travail. Pour effectuer un rendu, il vous suffit d'appeler la commande `submission`.

Dans le répertoire `submission` *vous trouverez : un fichier* `Makefile` *permettant de compiler votre code, un certain nombre de fichiers annexes, un fichier pour chaque question de la forme* `questionXX.c`*. Seuls les fichiers de question peuvent être modifiés, les autres seront remplacés lors de la correction par les originaux.*

Le `Makefile` *permet d'engendrer un petit programme de test pour chaque question. Le programme de test se charge des interactions avec l'utilisateur et appelle votre fonction avec les paramètres attendus. Pour obtenir le programme de test de chaque question il faut faire la commande :* `make questionXX`*.*

La compilation lors de la correction utilisera ce `Makefile`*, par conséquent vous n'aurez les points à la question XX que si la commande "*`make questionXX`*" réussit* ***et bien sûr que le résultat est correct****.*

Le barème est donné à titre indicatif et peut être sujet à modifications.

À la fin de ce document vous trouverez des annexes décrivants quelques consignes sur les programmes de tests.

Il y a 24 points et 17 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Question 1 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long my_intsqrt(unsigned long n);
```

`my_intsqrt(n)` calcule la racine carrée entière de n.
La racine carrée entière est la solution (entière) x à l'inéquation :

$$x^2 \leq n < (x+1)^2$$

Pour résoudre ce problème on utilise la méthode de Héron (une variante de la méthode de Newton). On considère une première approximation x de la racine (cette approximation doit être supérieur à la racine et inférieur à n, on prendra donc n comme première valeur). On calcule la prochaine approximation comme étant la moyenne arithmétique entre x et n/x et on continue tant que x est plus grand que n/x (c'est à dire tant que x est plus grand que la racine cherchée).

Exemple 1.1:

```
shell> ./question01 0 5
Fixed tests:
        my_intsqrt(    0)       =      0        [OK]
        my_intsqrt(    1)       =      1        [OK]
        my_intsqrt(    4)       =      2        [OK]
        my_intsqrt(   16)       =      4        [OK]
        my_intsqrt(   64)       =      8        [OK]
        my_intsqrt(  256)       =     16        [OK]
        my_intsqrt( 1024)       =     32        [OK]
        my_intsqrt( 4096)       =     64        [OK]
        my_intsqrt(16384)       =    128        [OK]
        my_intsqrt(65536)       =    256        [OK]
Random tests:
        my_intsqrt(1804289383)  =  42476        [OK]
        my_intsqrt( 846930886)  =  29102        [OK]
        my_intsqrt(1681692777)  =  41008        [OK]
        my_intsqrt(1714636915)  =  41408        [OK]
        my_intsqrt(1957747793)  =  44246        [OK]
```

Question 2 (1)

Écrire la(les) fonction(s) suivante(s):

```
void array_reverse(int *begin, int *end);
```

`array_reverse(begin, end)` inverse le contenu des cases du tableau entre `begin` (inclus) et `end` (exclus.)

Exemple 2.1:
```
shell> ./question02 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  after reverse:
    |   5 |   4 |   3 |   2 |   1 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  after reverse:
    | 793 | 915 | 777 | 886 | 383 |
shell> ./question02 0 6
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |
  after reverse:
    |   6 |   5 |   4 |   3 |   2 |   1 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 |
  after reverse:
    | 335 | 793 | 915 | 777 | 886 | 383 |
```

Question 3 (1)

Écrire la(les) fonction(s) suivante(s):

```c
void array_merge(int *dst, int *a1, int *a2, int *e1, int *e2);
```

`array_merge(dst, a1, a2, e1, e2)` fusionne les deux tableaux triés `a1` et `a2` dans le tableau `dst`. `e1` (respectivement `e2`) est le pointeur de fin (exclus) du tableau `a1` (respectivement `a2`). La zone mémoire pointée par `dst` est réputée suffisament grande pour contenir le contenu des deux tableaux (la fin du tableau `dst` se trouve à l'adresse `dst + e1 - a1 + e2 - a2`.)

Il n'y a aucune contrainte de taille entre `a1` et `a2`.

Le résultat de la fusion (dans `dst`) doit bien évidement être trié.

Exemple 3.1:
```
shell> ./question03 0 7
Fixed tests:
  array1:
    |   1 |   2 |   3 |
```

```
  array2:
    |   4 |   5 |   6 |   7 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Fixed tests (2):
  array1:
    |   1 |   3 |   5 |   7 |
  array2:
    |   2 |   4 |   6 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Random tests:
  array1:
    |  10 |  28 |  44 |
  array2:
    |  19 |  26 |  39 |  49 |
  array_merge(dst, array1, array2, ...)
  dst:
    |  10 |  19 |  26 |  28 |  39 |  44 |  49 |
```

Question 4 (1)

Écrire la(les) fonction(s) suivante(s):

```c
size_t mystrlen(char *s);
```

`mystrlen(s)` renvoie le nombre de caractères de la chaîne `s` (le pointeur ne sera pas `NULL`.) Vous devrez respecter le comportement attendu pour la fonction `strlen(3)`.

Exemple 4.1:

```
shell> ./question04 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 5 (1)

Écrire la(les) fonction(s) suivante(s):

```c
char *mystrncpy(char *dst, char *src, size_t len);
```

`mystrncpy(dst,src,len)` : copie au plus `len` caractères de la chaîne `src` dans la chaîne `dst`. On suppose que `src` et `dst` sont non `NULL`, `src` est terminée par un `'\0'` et `dst` est de taille suffisante.

Dans tous les cas, `mystrncpy(dst,src,len)` écrit exactement `len` caractères dans `dst`. Si le nombre de caractères de `src` est inférieur à `len`, alors votre fonction devra remplir la fin de `dst` par des `'\0'`. Sinon (`src` plus grand que `len`) votre fonction ne doit pas mettre de `'\0'` à la fin de `dst` (voir `strncpy(3)`.)

Je vous conseille fortement de lire la page de manuel de la fonction `strncpy(3)` qui fournit une description complète de cette fonction.

Exemple 5.1:

```
shell> ./question05 0 10
src = "n{6\Pavw[:"

test: mystrncpy(dst,src,11)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,5)
dst = "n{6\P"
-- check:
  first char:  [OK]
  last char:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,20)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,0)
-- check:
  overflow:  [OK]
```

Question 6 (1)

Écrire la(les) fonction(s) suivante(s):

```
str_toupper(char *s);
```

`str_toupper(s)` transforme toutes les lettres minuscules de la chaîne `s` (terminée par le charactère `'\0'`) en majuscule et laisse inchangés les autres charactères.
On rappel que les charactères en C ne sont que des entiers et que si c est une variable de type char contenant un lettre minuscule, alors c vérifie : `'a' <= c && c <= 'z'`. De même, si c est une lettre majuscule, alors : `'A' <= c && c <= 'Z'`.

Exemple 6.1:

```
shell> ./question06 0 26
Fixed tests:
  s = "abcdefghijklmnopqrstuvwxyz"
  str_to_upper(s)
  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Random tests:
  s = "n{6\Pavw[:mO4=ZvMD^bvU;e'R"
  str_to_upper(s)
  s = "N{6\PAVW[:MO4=ZVMD^BVU;E'R"
```

Question 7 (1)

Écrire la(les) fonction(s) suivante(s):

```
struct matrix *matrix_mul(struct matrix *A, struct matrix *B);
```

`matrix_mul(A, B)` calcule le produit des matrices `A` et `B`. Le résultat est une nouvelle matrice qui devra être correctement créée et allouée.
Les matrices sont représentée par le type suivant :

```
struct matrix {
  size_t lines, cols;
  int    *data;
};
```

De manière évidente, `lines` représente le nombre de lignes de la matrice et `cols` son nombre de colonnes. Le champ `data` est un pointeur sur une zone mémoire contenant `lines * cols` entiers correspondant aux cases de la matrice stockées en mode *ligne d'abord*.
Pour accéder à la case (i, j) de la matrice `A`, on utilisera le décallage classique : `A->data[i * A->cols + j]`.
On rappelle que le produit d'une matrice A de dimensions $n \times m$ par une matrice B de dimensions $m \times p$ est la matrice AB de dimension $(n \times p)$, dont les cases sont données par la formule suivante :

$$AB_{i,j} = \Sigma_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

Exemple 7.1:

```
shell> ./question07 0 3 5
Test with id matrix:
A =
|   83 |   86 |   77 |
|   15 |   93 |   35 |
|   86 |   92 |   49 |
id =
|    1 |    0 |    0 |
|    0 |    1 |    0 |
|    0 |    0 |    1 |
C = A * id
|   83 |   86 |   77 |
|   15 |   93 |   35 |
|   86 |   92 |   49 |
Random tests:
A =
|   21 |   62 |   27 |   90 |   59 |
|   63 |   26 |   40 |   26 |   72 |
|   36 |   11 |   68 |   67 |   29 |
B =
|   82 |   30 |   62 |
|   23 |   67 |   35 |
|   29 |    2 |   22 |
|   58 |   69 |   67 |
|   93 |   56 |   11 |
C = A * B
| 14638 | 14352 | 10745 |
| 15128 | 9538 | 8230 |
| 11760 | 8200 | 8921 |
```

Question 8 (1)

Écrire la(les) fonction(s) suivante(s):

```c
void list_insert(struct list *list, struct list *elm);
```

`list_insert(list,elm)` insert le maillon (déjà alloué) `elm` dans la liste chaînée `list` à sa place. La liste est triée en ordre croissant et dispose d'une sentinelle.
Le type représentant les listes est le suivant :

```c
struct list {
  struct list *next;
  int          val;
};
```

Exemple 8.1:
```
shell> ./question08 0 5
Fixed tests:
list = EMPTY
  insert 1 in list
  insert 2 in list
  insert 3 in list
  insert 4 in list
  insert 5 in list
list = ->   1 ->   2 ->   3 ->   4 ->   5
Random tests:
list = EMPTY
  insert 383 in list
  insert 886 in list
  insert 777 in list
  insert 915 in list
  insert 793 in list
list = -> 383 -> 777 -> 793 -> 886 -> 915
```

Question 9 (1)

Écrire la(les) fonction(s) suivante(s):

```c
void list_reverse(struct list *list);
```

`list_reverse(list)` renverse en place la liste `list`. Cette liste a une sentinelle qui doit être laissée en tête de la liste résultat.

Attention : il s'agit d'une liste intrusive (en gros vous n'avez pas accès au contenu) vous devez donc bien détacher les maillons de la liste et les racrocher en ordre inverse.

Le type représentant les listes est le suivant :

```c
struct list {
  struct list *next;
};
```

Exemple 9.1:
```
shell> ./question09 0 5
Fixed tests:
list = ->   1 ->   2 ->   3 ->   4 ->   5
list_reverse(list)
list = ->   5 ->   4 ->   3 ->   2 ->   1
```

```
Random tests:
list = -> 793 -> 915 -> 777 -> 886 -> 383
list_reverse(list)
list = -> 383 -> 886 -> 777 -> 915 -> 793
```

Programmation en Go

Pour cette partie, les squelettes de question se trouve dans le répertoire `GoPart/src` et vous disposez d'un sous répertoire par question.

Pour tester votre code (dans le répertoire de la question) il vous suffit de taper : `go test -v`. La sortie vous indiquera votre succès ou sinon quels tests ont échoués et affichera également quelques traces. Pour certaines questions ils sera possible de passer des paramètres suplémentaires dont vous pourrez voir les exemples pour chaque question.

Comme pour les questions de C, les tests disponibles en exam ne sont pas complets : **Le fait que le test renvoie OK n'indique pas que votre code réponde parfaitement à la question, vous devez inspecter les résultats affichés.**

Question 10 (1)

Écrire la(les) fonction(s) suivante(s):

```go
func Fact(n int) (r int)
```

`Fact(n)` calcule factorielle de n.

On rappelle la définition de factorielle :

$$\texttt{Fact}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n \times \texttt{Fact}(n-1) & \text{otherwise} \end{cases}$$

Répertoire de la question : `GoPart/src/question10`

Fichier de la question : `fact.go`

Exemple 10.1:

```
shell> go test
=== RUN   TestFact0
--- PASS: TestFact0 (0.00s)
=== RUN   TestFact1
--- PASS: TestFact1 (0.00s)
=== RUN   TestFactn
--- PASS: TestFactn (0.00s)
question10_test.go:23: Fact(2) = 2
question10_test.go:23: Fact(3) = 6
question10_test.go:23: Fact(4) = 24
question10_test.go:23: Fact(5) = 120
question10_test.go:23: Fact(6) = 720
question10_test.go:23: Fact(7) = 5040
question10_test.go:23: Fact(8) = 40320
question10_test.go:23: Fact(9) = 362880
question10_test.go:23: Fact(10) = 3628800
question10_test.go:23: Fact(11) = 39916800
question10_test.go:23: Fact(12) = 479001600
```

```
question10_test.go:23: Fact(13) = 6227020800
question10_test.go:23: Fact(14) = 87178291200
PASS
ok   question10 0.002s
```

Question 11 (2)

Écrire la(les) fonction(s) suivante(s):

```go
func Sort(v []int) (sorted []int)
```

`Sort(v)` trie le tableau et renvoie le tableau trié en utilisant l'algorithme de trie fusion (*merge sort*.)

Vous aurez besoin d'implémenter une fonction de fusion (*merge*) de deux vecteurs déjà triés.

On rappelle le principe du tri fusion :

```
merge(v1, v2): returned a sorted merge of v1 and v2

Sort(v):
  si longueur de v inférieure à 2:
    renvoyer une copie de v
  sinon:
    mid <- longueur de v divisée par 2
    renvoyer merge(Sort(v de 0 à mid), Sort(v de mid à fin de v))
```

Pour information, voici quelques constructions go qui pourraient être utiles :

```go
// v a vector
// copy v in new vector r
r := make([]int, len(v))
copy(r, v)

// get a subrange from left to right of v
vsub := v[left:right]
// special cases
vsub := v[:right]
vsub := v[left:]
```

Répertoire de la question : `GoPart/src/question11`

Fichier de la question : `merge_sort.go`

Exemple 11.1:

```
shell> go test -v -args 4
```

```
=== RUN   TestSorted
--- PASS: TestSorted (0.00s)
question11_test.go:47: After sort:  [0 1 2 3]
=== RUN   TestRevSorted
--- PASS: TestRevSorted (0.00s)
question11_test.go:59: After sort:  [0 1 2 3]
=== RUN   TestRandom
--- PASS: TestRandom (0.00s)
question11_test.go:72: After sort:  [11911 19767 32023 51502]
PASS
ok   question11 0.003s
```

Question 12 (3)

Écrire la(les) fonction(s) suivante(s):

```go
func next_permutation(v []int) (next bool)
```

`next_permutation(v)` permute les éléments de v en respectant l'ordre lexicographique et renvoie faux lorsque l'on revient à la première permutation et vrai sinon. Toutes les valeurs du tableau sont distincts.

L'ordre lexicographique correspond à un ordre de type dictionnaire : on compare les éléments depuis le début, tant qu'ils sont égaux on avance dans les deux tableaux et lorsque les éléments courrants des deux tableaux sont différents on compare ces éléments. C'est exact le même ordre que pour les mots du dictionnaire ou l'ordre sur les chaînes. Par exemple, pour un tableau de 3 éléments, l'ordre lexicographique nous donne la séquence de permutations suivante :

— Tableau initial : $1, 2, 3$
— Première permutation : $1, 3, 2$
— Suivante : $2, 1, 3$
— Suivante : $2, 3, 1$
— Suivante : $3, 1, 2$
— Dernière : $3, 2, 1$

La fonction `next_permutation(v)` remplace v par la prochaine permutation dans l'ordre lexicographique.

L'algorithme est le suivant :

```
next_permutation(v):
  next = premier élément en partant de la fin qui n'est pas plus grand que son
         voisin de droite
  si le tableau est trié en ordre inverse, l'inverser et renvoyer faux
  repl = position tel que v[next] < v[repl] et pour tout i dans
         [next+1..repl], v[i] > v[repl]
  echange v[next] et v[repl]
  inverser v entre next + 1 et la fin de v
  renvoyer vrai
```

Par exemple, on part du tableau : 1, 2, 5, 4, 3. La valeur de `next` est 1 (`v[next] == 2`), repl vaudra 4 (le dernier élément), on échange et on obtient 1, 3, 5, 4, 2 et enfin on inverse pour obtenir la prochaine permutation : 1, 3, 2, 4, 5.
Remarque : il peut être judicieux de faire la prochaine question avant ...

Répertoire de la question : `GoPart/src/question12`

Fichier de la question : `next_permutation.go`

Exemple 12.1:

```
shell> go test -v
=== RUN   TestFixedSmallSize
--- PASS: TestFixedSmallSize (0.00s)
question12_test.go:68: Testing with a size of 3:
question12_test.go:72: Original vector: [1 2 3]
question12_test.go:74: Next_permutation(v): [1 3 2]
question12_test.go:74: Next_permutation(v): [2 1 3]
question12_test.go:74: Next_permutation(v): [2 3 1]
question12_test.go:74: Next_permutation(v): [3 1 2]
question12_test.go:74: Next_permutation(v): [3 2 1]
question12_test.go:81: last permutation(v): [1 2 3]
=== RUN   TestBiggerSize
--- PASS: TestBiggerSize (0.00s)
question12_test.go:97: Testing with a size of 4:
question12_test.go:104: Original vector: [1 2 3 4]
question12_test.go:108: Next_permutation(v): [1 2 4 3]
question12_test.go:108: Next_permutation(v): [1 3 2 4]
question12_test.go:108: Next_permutation(v): [1 3 4 2]
question12_test.go:108: Next_permutation(v): [1 4 2 3]
question12_test.go:108: Next_permutation(v): [1 4 3 2]
question12_test.go:108: Next_permutation(v): [2 1 3 4]
question12_test.go:108: Next_permutation(v): [2 1 4 3]
question12_test.go:108: Next_permutation(v): [2 3 1 4]
question12_test.go:108: Next_permutation(v): [2 3 4 1]
question12_test.go:108: Next_permutation(v): [2 4 1 3]
question12_test.go:108: Next_permutation(v): [2 4 3 1]
question12_test.go:108: Next_permutation(v): [3 1 2 4]
question12_test.go:108: Next_permutation(v): [3 1 4 2]
question12_test.go:108: Next_permutation(v): [3 2 1 4]
question12_test.go:108: Next_permutation(v): [3 2 4 1]
question12_test.go:108: Next_permutation(v): [3 4 1 2]
question12_test.go:108: Next_permutation(v): [3 4 2 1]
question12_test.go:108: Next_permutation(v): [4 1 2 3]
question12_test.go:108: Next_permutation(v): [4 1 3 2]
question12_test.go:108: Next_permutation(v): [4 2 1 3]
question12_test.go:108: Next_permutation(v): [4 2 3 1]
question12_test.go:108: Next_permutation(v): [4 3 1 2]
question12_test.go:108: Next_permutation(v): [4 3 2 1]
question12_test.go:116: last permutation(v): [1 2 3 4]
question12_test.go:120: Number of permutation: 24
PASS
ok   question12 0.012s
```

Question 13 (1)

Écrire la(les) fonction(s) suivante(s):

```go
func Reverse(v []int)
```

`Reverse(v)` inverse le tableau v

Répertoire de la question : `GoPart/src/question13`

Fichier de la question : `reverse.go`

Exemple 13.1:

```
shell> go test -v -args 4
=== RUN   TestBasic
--- PASS: TestBasic (0.00s)
question13_test.go:39: Before reverse: [0 1 2 3 4]
question13_test.go:41: After reverse:  [4 3 2 1 0]
=== RUN   TestRandomFixedLen
--- PASS: TestRandomFixedLen (0.00s)
question13_test.go:56: Before reverse: [5577006791947779410 8674665223082153551 6129484611666145821 4037200794235010051]
question13_test.go:58: After reverse:  [4037200794235010051 6129484611666145821 8674665223082153551 5577006791947779410]
=== RUN   TestFullRandom
--- PASS: TestFullRandom (0.00s)
question13_test.go:74: Before reverse: [6334824724549167320 605394647632969758]
question13_test.go:76: After reverse:  [605394647632969758 6334824724549167320]
PASS
ok   question13 0.007s
```

Question 14 (1)

Écrire la(les) fonction(s) suivante(s):

```go
func Fibo_list(length int) (fl []int)
```

`Fibo_list(length)` construit la liste des nombres de la suite de Fibonacci de longueur `length`.
On rappelle la définition de la suite de Fibonacci :

$$Fibo(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ Fibo(n-1) + Fibo(n-2) & \text{sinon} \end{cases}$$

Le paramètre donne la longueur attendue de la liste, donc :
— Vous devez renvoyer une *liste* vide pour `length == 0`
— `length == 1` renvoie une liste ne contenant que la valeur 1
— `length == 2` renvoie les deux éléments initiaux de la suite 1 et 1.
La liste est construite en utilisant les slices de Go, on rappelle les syntaxe suivante :

```go
// An empty list
var l []int = nil

// A list of one element, like [1]
l := []int{1}

// A list of two elements, like [1,1]
l := []int{1,1}
```

Répertoire de la question : `GoPart/src/question14`

Fichier de la question : `fibo_list.go`

Exemple 14.1:

```
shell> go test -v -args 5
=== RUN   TestLen0
--- PASS: TestLen0 (0.00s)
question14_test.go:35: Fibo_list(0) = []
=== RUN   TestLen1
--- PASS: TestLen1 (0.00s)
question14_test.go:43: Fibo_list(1) = [1]
=== RUN   TestRandLen
--- PASS: TestRandLen (0.00s)
question14_test.go:62: Fibo_list(5) = [1 1 2 3 5]
PASS
ok    question14 0.001s
```

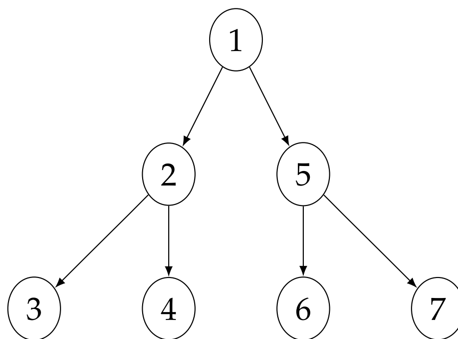

Figure 1 – Exemple d'arbre

Question 15 (1)

Écrire la(les) fonction(s) suivante(s):

```go
func (t *Tree) PreOrderList() (l []int)
```

`t.PreOrderList()` construit la liste des clefs de l'arbre `t` dans l'ordre d'un parcours profondeur (DFS) en rencontre préfixe.

Les arbres binaires sont représentés par le type suivant :

```go
type Tree struct {
        Key         int
        Left, Right *Tree
}
```

Pour simplifier la vérification des tests pendant l'épreuve (et uniquement pendant l'épreuve) les arbres construits pour les tests sont des arbres complets (*perfect tree*, toutes les feuilles au même niveau et tous les nœuds internes ont deux fils) dont les clefs sont des entiers assignés dans l'ordre de rencontre préfixe d'un parcours profondeur (voir la figure 1).

Répertoire de la question : `GoPart/src/question15`

Fichier de la question : `tree_dfs.go`

Exemple 15.1:

```
shell> go test -v -args 2
=== RUN   TestTrivialTree0
--- PASS: TestTrivialTree0 (0.00s)
question15_test.go:59: Building regular tree of depth -1
question15_test.go:61: tree.PreOrderList() =  []
=== RUN   TestTrivialTree1
--- PASS: TestTrivialTree1 (0.00s)
question15_test.go:70: Building regular tree of depth 0
question15_test.go:72: tree.PreOrderList() =  [1]
=== RUN   TestFixedDepthRegularTree
--- PASS: TestFixedDepthRegularTree (0.00s)
question15_test.go:81: Building regular tree of depth 2
question15_test.go:83: tree.PreOrderList() =  [1 2 3 4 5 6 7]
PASS
ok    question15 0.001s
```

Graphes

Les deux derniers exercices s'appuient utilisent des graphes orientés, représenté par les types suivants :

```go
type Vertex struct {
        Succ, Pred []int
}

type Graph struct {
        Order    int
        Vertices []Vertex
}
```

C'est une représentation semi-statique par listes d'adjacence. Le type `Graph` contient le nombre de sommet (`Order`) et un tableau de sommets (`Vertices`) ou l'on trouve le sommet `i` à la case `i` (les sommets sont numérotés de `0` à `Order - 1`.

Chaque sommet (type `Vertex`) dispose ensuite d'un tableau de successeurs `Succ` contenant l'identifiant de chaqu'un des successeurs du sommet courrant.

Il y a des fichiers représentant les graphes (extension `.nde`) dans chaque répertoire de question. Le format est simple : première ligne le nombre de sommet, et ensuite une ligne par arcs.

Si vous désirez visualiser les graphes, le répertoire `tools` contient un petit programme go qui permet de transformer un fichier de graphe en fichier au format dot (qui pourra être transformé en image après.)

Voici un petit exemple de session pour voir :

```
> cd ~/submissions/GoPart/src/tools
> go build
> ./tools -graph ../question16/graph.nde -dot q16-graph.dot
> dot -Tpng q16-graph.dot -O
> display q16-graph.dot.png
...
```

Question 16 (3)

Écrire la(les) fonction(s) suivante(s):

```go
func (g *Graph) DFSBuildSpanningTree(v int, st *SpanTree)
```

`g.DFSBuildSpanningTree(v, st)` traverse le graphe orienté `g` et construit l'arbre couvrant correspondant au parcours profondeur complet (tous les sommets doivent être atteints) avec `v` comme sommet initial.

La construction de l'arbre couvrant (*spanning tree*) utilise le type `SpanTree` (décrit dans le fichier `spanning_tree.go`) qui fournit les opérations suivantes :

```go
func (st *SpanTree) AddTreeEdge(x, y int)

func (st *SpanTree) AddBackEdge(x, y int)

func (st *SpanTree) AddForwardEdge(x, y int)

func (st *SpanTree) AddCrossEdge(x, y int)
```

Pendant la traversé du graphe, vous devrez ajouter les arcs de l'arbre (forêt) couvrant(e), ainsi que les arcs supplémentaires habituels : arcs en arrière, arcs en avant et arcs croisés.

On rappelle que les tests suivants sont suffisants pour établir le type d'arcs :

— Les arcs couvrants (*Tree Edge*) sont ceux utilisés pour le parcours ;
— Les types suivants correspondent tous au cas de sommets déjà marqués :
  —
  — arcs en arrière (*back edge*) $u \to v$: `post[v] == 0` (ordre suffixe non défini) ;
  — arcs en avant (*forward edges*) $u \to v$: `pre[u] < pre[v]` ;
  — arcs croisés (*cross edges*) les arcs restants.

Si vous désirez visualiser l'arbre de recouvrement, le programme de tests propose une option `-dot dotfile.dot` qui permet de générer un fichier que vous pourrez visualiser en générant le l'image correspondante.

Répertoire de la question : `GoPart/src/question16`

Fichier de la question : `dfs_graph.go`

Exemple 16.1:

```
shell> go test -v
=== RUN   TestGraph
--- PASS: TestGraph (0.00s)
spanning_tree.go:52: Edges: [{0 1} {1 3} {3 4} {4 5} {0 2}]
spanning_tree.go:53: BackEdges: []
spanning_tree.go:54: ForwardEdges: [{3 5} {1 5}]
spanning_tree.go:55: CrossEdges: [{2 4}]
PASS
ok    question16 0.001s
shell> go test -v -dot graph2.dot -graph graph2.nde
=== RUN   TestGraph
--- PASS: TestGraph (0.00s)
spanning_tree.go:52: Edges: [{0 1} {1 3} {3 5} {5 4} {4 6} {6 7} {0 2} {0 8} {8 9}]
spanning_tree.go:53: BackEdges: [{7 3}]
spanning_tree.go:54: ForwardEdges: [{5 7} {0 5}]
spanning_tree.go:55: CrossEdges: [{2 1} {8 5}]
PASS
ok    question16 0.002s
shell> dot -Tpng graph2.dot -O
shell> display graph2.dot.png
```

Question 17 (3)

Écrire la(les) fonction(s) suivante(s):

```go
func (g *Graph) BFS_dist(src int, dist []int)
```

`g.BFS_dist(src, dist)` traverse le graphe orienté `g` via un parcours en largeur (*BFS*) depuis le sommet `src` et calcule les distances vers tous les sommets atteignables de `g`.

Pour l'implémentation de la file, vous pouvez utiliser les slices : append ajoute en fin, le sommet de la file est en case `0` et en utilisant les sous-slices, vous pouvez *enlever* la première case.

```go
// empty queue
var q []int

// push 0 in the queue
q = append(q, 0)

// queue is not empty ?
len(q) > 0

// get next element and pop it
```

```
x := q[0]
q = q[1:]
```

Répertoire de la question : `GoPart/src/question17`

Fichier de la question : `bfs_graph.go`

Exemple 17.1:

```
shell>  go test -v
=== RUN   TestGraph
--- PASS: TestGraph (0.00s)
question17_test.go:22: [0 1 1 2 2 2]
PASS
ok    question17 0.001s
shell> go test -v -graph graph2.nde
=== RUN   TestGraph
--- PASS: TestGraph (0.00s)
question17_test.go:22: [0 1 1 2 2 1 3 2 1 2]
PASS
ok    question17 0.001s
```

Remarques sur la session exam

À la fin de l'épreuve, vous devez quitter votre session en fermant l'horloge (et uniquement en fermant celle-ci.) Dans tous les cas, votre session sera également fermée à la fin du temps imparti.

Lorsque l'horloge est fermée (volontairement, ou en fin de session), la console qui vous a demandé le token vous demande votre mot de passe. **Vous devez rentrer votre mot de passe UNIX pour que le rendu ai bien lieu.**

Vous pouvez aussi rendre sans quitter : il y a une commande (dans le shell) appellée `submission` qui déclenche un rendu comme si vous quittiez (demande de mot de passe et envoie du rendu.)

Après la fin du test (dans les minutes qui suivent) vous pouvez redémarrer votre machine et vous connecter pour re-rendre (si par exemple il y a eu une erreur pendant votre premier rendu.)

Remarques sur les squelettes de questions

Pour chaque question, un embryon de code vous est fourni. Ce code correspond au strict minimum pour que la question compile et que l'appel au programme de test échoue avec une erreur identifiable. Par conséquent, vous devez supprimer du corps des fonctions à écrire, le code provoquant l'erreur, sous peine de voir votre programme échouer lors des tests (ne laissez surtout pas la ligne `REMOVE_ME(...)`.)

Exemple 1:
À titre d'exemple, si l'on vous demande la fonction C suivante :

```
int identity(int x);
```

`identity(x)` *renvoi* `x`.
Vous trouverez dans le fichier de question correspondant le code :

```
int identity(int x) {
  /* FIX ME */
  REMOVE_ME(x);
}
```

Que vous devrez remplacer par :

```
int identity(int x) {
  return x;
}
```

Remarques sur les programmes de test

La commande `make questionXX` produira un binaire `questionXX`. Ce binaire peut être utilisé pour tester votre réponse à la question X du sujet. Tous les binaires attendent des paramètres et affichent une aide succincte s'il sont appelés avec l'option `-help`.

Exemple 2:
Le binaire produit pour la question 1 (il s'agit d'un exemple qui ne correspond pas forcement au sujet)

```
Question 1:
./question01 graine taille
  -help   Display this list of options
  --help  Display this list of options
```

Les deux paramètres sont donc `graine` et `taille`. Les paramètres nommés `graine` (présent pour chaque question, ou presque) font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question03 X Y` va initialiser le générateur de nombre aléatoire avec X (ici, Y servant de taille à la liste générée.) Pour les mêmes valeurs de X on obtiendra les mêmes données (la séquence engendrée par une graine donnée est toujours la même.)

Le reste du temps les noms des paramètres donnés dans l'aide sont explicites (par rapport au sujet.) Si vous avez une doute sur l'utilisation vous pouvez jeter un oeil sur fichier contenant les tests de la question X (`test_qXX.c`.)