Programmation Avancée (INF441)
Contrôle de remplacement et de rattrapage
CORRIGÉ

3 septembre 2015

Solution de la question 1 Une solution est donnée dans la figure 1. Comme proposé par le sujet, on utilise un vecteur `levels`, de taille `P`, tel que `levels.get(p)` est l'ensemble des éléments de priorité `p`. Cet ensemble est lui-même représenté par un vecteur (de taille arbitraire), ce qui fait que le type du champ `levels` est `Vector<Vector<E>>`.

On a choisi ici de ne pas stocker la borne `P` dans un champ ; on aurait pu. On peut obtenir `P`, lorsque cela est nécessaire, via l'expression `levels.size()`.

Le constructeur alloue le vecteur `levels`, lui donne la taille `P`. Après ces deux instructions, chaque case du vecteur `levels` contient `null`. Or, on souhaite que chaque case contienne un ensemble (représenté par un vecteur) vide. Il faut donc encore (à l'aide d'une boucle) initialiser chaque case.

La méthode `insert` fait l'hypothèse que la priorité `p` est comprise entre 0 (inclus) et `P` (exclus). La première ligne vérifie cette hypothèse : ce style « défensif » est une bonne pratique, mais n'est pas obligatoire. La seconde ligne ajoute l'élément `element` à l'ensemble `levels.get(p)` des éléments de priorité `p`.

La méthode `extract` doit extraire un élément de priorité minimale. Or, la représentation mémoire proposée par le sujet ne permet pas de connaître immédiatement à quel niveau se situe cet élément. Il faut donc examiner tous les niveaux, l'un après l'autre. Dès qu'on trouve un niveau non vide, on en extrait un élément, que l'on renvoie. Si on ne trouve aucun niveau non vide, alors la file de priorité est vide : on lance alors l'exception `NoSuchElementException`.

La boucle qui énumère les niveaux est écrite ici sous la forme d'une boucle « foreach » :

```
for (Vector<E> level : levels) {
  ...
}
```

On pouvait aussi utiliser la forme plus classique :

```
int P = levels.size();
for (int i = 0; i < P; i++) {
  Vector<E> level = levels.get(i);
  ...
}
```

Notons que, lorsque l'on trouve un niveau non vide, on peut en principe en extraire n'importe quel élément. Nous en extrayons ici le dernier élément, via l'appel `level.remove(size - 1)`, qui a un coût $O(1)$. On pourrait en extraire le premier élément, via l'appel `level.remove(0)`. Toutefois, cet appel a un coût $O(k)$, où k est la taille du vecteur `level`, parce que la méthode `remove` décale d'un cran tous les éléments qui suivent celui que l'on retire. Ce ne serait donc pas une bonne chose. □

```
public class VectorPriorityQueue<E> {

  // This vector has size P. At index p in this vector,
  // we find a set of elements whose priority is p.
  private final Vector<Vector<E>> levels;

  public VectorPriorityQueue (int P) {
    levels = new Vector<Vector<E>> ();
    levels.setSize(P);
    for (int p = 0; p < P; p++)
      levels.set(p, new Vector<E> ());
  }

  public void insert (int p, E element) {
    assert (0 <= p && p < levels.size()); // optional
    levels.get(p).add(element);
  }

  public E extract () throws NoSuchElementException {
    for (Vector<E> level : levels) {
      int size = level.size();
      if (size > 0)
        return level.remove(size - 1);
    }
    throw new NoSuchElementException ();
  }

}
```

FIGURE 1 – La classe `VectorPriorityQueue`, sous sa forme la plus simple

Solution de la question 2 Examinons d'abord le constructeur. L'allocation d'un vecteur coûte $O(1)$ seulement, puisque sa taille est initialement nulle. Toutefois, l'appel `levels.setSize(P)` coûte $O(P)$, puisque cet appel initialise chaque case du vecteur à la valeur `null`. Enfin, la boucle `for (int p = 0; p < P; p++) { ... }` coûte également $O(P)$, puisque son corps ne contient que des opérations de complexité $O(1)$. La complexité du constructeur est donc $O(P)$.

La complexité de la méthode `insert` est $O(1)$. En effet, l'appel `levels.get(p)` a un coût constant – c'est un simple accès en lecture à un tableau – et l'appel à la méthode `add` a également un coût constant – nous l'avons admis.

La complexité de la méthode `extract` est $O(P)$. En effet, la boucle `for` est exécutée, dans le pire cas, P fois, et son corps ne contient que des opérations de complexité $O(1)$. Cette méthode est le point faible de cette structure de données : en pratique, à moins que P soit très petit, ce coût risque d'être élevé. □

Solution de la question 3 La solution est simple :

```
static void sort (String[] strings, int P) {
  VectorPriorityQueue<String> q =
    new VectorPriorityQueue<> (P);
  for (String s : strings)
    q.insert(s.length, s);
  for (int i = 0; i < strings.length; i++)
    strings[i] = q.extract();
}
```

On crée d'abord une nouvelle file de priorité, initialement vide, dont les priorités sont bornées par P. Puis, pour chaque chaîne de caractères `s` issue du tableau `strings`, on ajoute `s` à la file, avec pour priorité `s.length`, puisque l'on souhaite trier les chaînes de caractères par longueur croissante. Enfin, on extrait un par un les éléments de la file, en les recopiant au fur et à mesure dans le tableau `strings`.

Il ne s'agit ici que d'un exemple-jouet. Toutefois, nous ne sommes pas très éloignés d'un véritable algorithme de tri, le « tri comptage » ou *counting sort*, qui est utilisé lorsque la clef de tri est un entier borné, comme c'est le cas ici. □

Solution de la question 4 D'après la question 2, la création de la file de priorité a un coût $O(P)$. La boucle d'insertion des éléments est exécutée n fois et son corps a un coût $O(1)$. Son coût total est donc $O(n)$. Enfin, la boucle d'extraction des éléments est exécutée n fois et son corps a un coût $O(P)$. Son coût total est donc $O(nP)$. Le coût de l'ensemble est donc $O(P + n + nP)$, c'est-à-dire $O(nP)$. □

Solution de la question 5 Notons d'abord que, si nous n'apportons aucune modification à la classe `VectorPriorityQueue`, le coût total de n insertions et n extractions est $O(nP)$. C'est l'analyse effectuée en réponse à la question précédente.

Pour faire mieux, il faut diminuer le coût de la boucle située dans la méthode `extract`. Sous l'hypothèse que la priorité des éléments nouvellement insérés est au moins égale à la priorité du dernier élément extrait, c'est possible. L'idée est simple : au lieu de parcourir à chaque fois tout le tableau `levels`, la recherche doit se faire à partir de l'indice `current`, où `current` est la priorité du dernier élément extrait.

On ajoute donc un champ `current` de type `int` (figure 2). Pour bien comprendre quel est le rôle de ce champ, on peut énoncer l'invariant suivant : à tout moment, si `p` est strictement inférieur à `current`, alors l'ensemble `levels.get(p)` est vide. En d'autres termes, la file ne contient aucun élément de priorité strictement inférieure à `current`.

On initialise `current` à la valeur 0. L'invariant est donc initialement trivialement satisfait.

```
public class VectorPriorityQueue<E> {

  // This vector has size P. At index p in this vector,
  // we find a set of elements whose priority is p.
  private final Vector<Vector<E>> levels;

  // Invariant: all levels strictly below "current" are empty.
  private int current;

  public VectorPriorityQueue (int P) {
    levels = new Vector<Vector<E>> ();
    levels.setSize(P);
    for (int p = 0; p < P; p++)
      levels.set(p, new Vector<E> ());
    current = 0;
  }

  public void insert (int p, E element) {
    assert (0 <= p && p < levels.size()); // optional
    assert (current <= p);                // optional
    levels.get(p).add(element);
  }

  public E extract () throws NoSuchElementException {
    int P = levels.size();
    while (current < P) {
      Vector<E> level = levels.get(current);
      int size = level.size();
      if (size > 0)
        return level.remove(size - 1);
      current++;
    }
    throw new NoSuchElementException ();
  }

}
```

FIGURE 2 – La classe `VectorPriorityQueue`, sous forme plus efficace

Dans `insert`, rien ne change. On peut toutefois, si l'on apprécie le style « défensif », vérifier que la propriété `current <= p` est satisfaite. Si elle est violée, cela signifie que l'utilisateur ne respecte pas le scénario qui a été énoncé. Si cette propriété est satisfaite, alors on voit clairement que l'invariant est maintenu.

Enfin, dans `extract`, au lieu d'énumérer tous les éléments du vecteur `levels`, on commence la recherche à partir de l'indice `current`. Cela s'écrit, par exemple, avec une boucle **`while`**. On pourrait aussi employer une boucle **`for`** un peu inhabituelle :

```
for (; current < P; current++) {
  ...
}
```

Examinons la complexité en temps de `extract`, sous cette forme. Dans le cas favorable où on trouve immédiatement un élément situé au niveau `current`, cette complexité est $O(1)$. Dans le cas défavorable où on doit passer au niveau suivant, on doit payer $O(1)$ puis recommencer. Heureusement, puisque `current` ne fait que croître avec le temps, ce cas défavorable ne se présente au pire que P fois **en tout**, durant toute la vie de la file de priorité. Le coût total de n insertions et n extractions est donc bien $O(n + P)$.

Pour finir, notons que dans `insert`, au lieu de `assert (current <= p);`, on pourrait écrire :

```
if (!(current <= p))
  current = p;
```

Ainsi, même si le scénario prévu n'est pas respecté par l'utilisateur, on fait en sorte de maintenir l'invariant. La file se comporte alors de façon correcte, quoi que fasse l'utilisateur. S'il respecte le scénario, alors le coût total de n insertions et n extractions sera $O(n + P)$. S'il ne le respecte pas, ce coût pourra être $O(nP)$. □

Solution de la question 6 Puisque `levels` est déjà un vecteur, c'est-à-dire un tableau redimensionnable, il est relativement facile d'aménager le code pour que P ne soit pas fixé lors de la construction de la file de priorité.

La figure 3 donne une solution. Elle est combinée ici avec les améliorations apportées lors de la question 5, mais cela n'était pas demandé.

Le constructeur n'attend plus d'argument. L'appel `levels.setSize(P)` disparaît, ainsi que la boucle d'initialisation qui suivait. Le tableau `levels` a donc initialement une taille nulle.

Le code d'initialisation qui se trouvait dans le constructeur est déplacé dans la méthode `insert`. Si la propriété `p < P` n'est pas satisfaite, alors il faut redimensionner (agrandir) le tableau `levels`. Sa taille doit être (au moins) `p + 1`. On la modifie donc via un appel à `setSize`, et on n'oublie pas d'initialiser les nouvelles cases du tableau `levels`, qui juste après l'appel à `setSize` contiennent **`null`**, en stockant dans chacune d'elles un (nouvel) ensemble vide d'éléments.

La méthode `extract` est inchangée. □

Solution de la question 7 Il s'agit de transcrire la définition mathématique, en distinguant d'abord les deux cas de base du cas général, et en utilisant deux appels récursifs dans ce dernier cas.

```
let rec c (n, p) =
  assert (0 <= p && p <= n); (* optional *)
  if p = 0 || p = n then 1
  else begin
    assert (0 < p && p < n); (* optional *)
    c (n - 1, p - 1) + c (n - 1, p)
  end
```

```
public class VectorPriorityQueue<E> {

  // At index p in this vector, we find a set of elements
  // whose priority is p.
  private final Vector<Vector<E>> levels;

  // Invariant: all levels strictly below "current" are empty.
  private int current;

  public VectorPriorityQueue () {
    levels = new Vector<Vector<E>> ();
    current = 0;
  }

  public void insert (int p, E element) {
    assert (0 <= p);                        // optional
    if (!(current <= p))
      current = p;
    int P = levels.size();
    if (!(p < P)) {
      levels.setSize(p + 1);
      for (int i = P; i <= p; i++)
        levels.set(i, new Vector<E> ());
    }
    levels.get(p).add(element);
  }

  public E extract () throws NoSuchElementException {
    int P = levels.size();
    while (current < P) {
      Vector<E> level = levels.get(current);
      int size = level.size();
      if (size > 0)
        return level.remove(size - 1);
      current++;
    }
    throw new NoSuchElementException ();
  }

}
```

FIGURE 3 – La classe `VectorPriorityQueue`, où l'on ne fixe pas P lors de la construction

Nous avons inclus deux assertions, qui ne sont pas nécessaires en principe, mais servent de documentation et facilitent le test : si on s'est trompé dans les conditions de bord, l'erreur sera probablement détectée. □

Solution de la question 8 Un appel à `c (n, _)` peut donner lieu à deux appels récursifs à `c (n - 1, _)`. La complexité est donc $O(2^n)$. □

Solution de la question 9 Comme l'affirme l'énoncé, la première chose à faire est d'allouer une nouvelle table de hachage, `table`, initialement vide. Cette table sera ensuite progressivement peuplée d'associations entre une clef `x` et une valeur `y`, où `y` est égal à `f x`.

```
let memo f =
  let table = Hashtbl.create 128 in
  let memo_f x =
    try
      Hashtbl.find table x
    with Not_found ->
      let y = f x in
      Hashtbl.add table x y;
      y
  in
  memo_f
```

Une fois la table créée, on définit une fonction `memo_f`, une version mémoïsante de la fonction `f`, puis on renvoie `memo_f`.

La fonction `memo_f` est définie comme suit. Lorsqu'on l'appelle avec un argument `x`, elle teste si `x` apparaît déjà dans la table. Si oui, elle renvoie immédiatement la valeur associée : on évite ainsi d'appeler `f` et de répéter le calcul. Si non, alors il faut appeler `f x` pour obtenir un résultat `y`. On renvoie alors `y`, après avoir mémorisé l'association entre `x` et `y` dans la table de hachage, afin que les prochains appels puissent bénéficier de cette information. □

Solution de la question 10 L'appel à `memo` initialise une table de hachage et construit une clôture. Le coût de l'opération `Hashtbl.create` ne dépend que de la taille initiale choisie pour la table, qui ici est une constante. Allouer une clôture a également un coût $O(1)$. Le coût de l'appel à `memo` est donc $O(1)$.

Le premier appel à `memo_c (n, p)` provoque un appel à `c (n, p)`, dont le coût est $O(2^n)$. Les opérations `Hashtbl.find` et `Hashtbl.add` ont un coût constant. Le coût total de cet appel est donc $O(2^n)$.

Le second appel à `memo_c (n, p)` ne provoque pas d'appel à la fonction `c`, puisque le résultat du calcul se trouve déjà dans la table. Son coût est donc $O(1)$. □

Solution de la question 11 Le résultat obtenu n'est pas vraiment satisfaisant. Lorsqu'on appelle `memo_c (n, p)`, si on la chance que le résultat soit déjà connu, alors on l'obtient en temps constant ; mais dans le cas contraire, le coût du calcul reste exponentiel. Cela provient du fait que les appels récursifs de la fonction `c` à elle-même ne bénéficient pas de la mémoïsation. Or, il y aurait beaucoup à gagner à l'exploiter, car de nombreux appels récursifs redondants sont effectués.

On peut définir une fonction `memo_c` plus efficace en ajoutant à la définition récursive de la fonction `c` un mécanisme de mémoïsation analogue à celui utilisé pour définir `memo` :

```
let table =
  Hashtbl.create 128
let rec memo_c (n, p) =
  try
    Hashtbl.find table (n, p)
  with Not_found ->
```

```
    let y =
      if p = 0 || p = n then 1
      else memo_c (n - 1, p - 1) + memo_c (n - 1, p)
    in
    Hashtbl.add table (n, p) y;
    y
```

La différence fondamentale vis-à-vis de la définition `let memo_c = memo c` de la question 12 est que les appels récursifs ne sont pas des appels à `c`, mais des appels à `memo_c`, donc bénéficient eux aussi de la mémoïsation.

Étudions de façon rigoureuse la complexité de ce code.

Un appel à `memo_c (n, p)` entraîne un certain nombre d'appels récursifs à `memo_c (n', p')` pour des valeurs de `n'` et `p'` qui satisfont `p' <= n' <= n`. Disons qu'un appel à `memo_c (n', p')` est « non trivial » lorsqu'il a lieu pour la première fois, et « trivial » les fois suivantes.

Un appel trivial a clairement un coût $O(1)$, puisqu'il termine après une consultation de la table de hachage. Imputons ce coût à son appelant (qui doit être un appel non trivial).

Un appel non trivial a aussi un coût $O(1)$, si l'on exclut le coût des appels récursifs non triviaux qu'il effectue. En effet, il n'effectue que des opérations de coût $O(1)$: opérations sur la table de hachage, opérations arithmétiques, tests, et appels récursifs triviaux.

En résumé, puisque le coût des appels triviaux a été imputé à leur appelant, le coût total est la somme des coûts des appels non triviaux. Puisque le coût individuel de ces appels est $O(1)$, leur coût total est, à une constante près, le nombre d'appels non triviaux.

Or, le nombre d'appels non triviaux est (au pire) le nombre de paires `(n', p')` qui satisfont `p' <= n' <= n`, c'est-à-dire $O(n^2)$.

La complexité en temps de `memo_c` est donc $O(n^2)$.

La complexité en espace est la même, d'ailleurs, puisqu'il y a autant d'éléments dans la table de hachage, à la fin, que d'appels non triviaux.

Ceci conclut la réponse à cette question.

En guise d'ultime remarque, notons que, ci-dessus, on a mélangé le code de calcul des coefficients binômiaux et le code de mémoïsation. C'est dommage : dans les questions 11 et 12, on avait réussi à les séparer. On peut les isoler à nouveau de la façon suivante :

```
let c self (n, p) =
  if p = 0 || p = n then 1
  else self (n - 1, p - 1) + self (n - 1, p)

let memo_rec f =
  let table = Hashtbl.create 128 in
  let rec memo_f x =
    try
      Hashtbl.find table x
    with Not_found ->
      let y = f memo_f x in
      Hashtbl.add table x y;
      y
  in
  memo_f

let memo_c =
  memo_rec c
```

Ici, la fonction `c` n'est pas récursive. Au lieu de s'appeler directement elle-même (ce qui serait mauvais, car cet appel récursif ne bénéficierait pas de la mémoisation), elle appelle une fonction `self` qu'elle a reçu en argument. On dit parfois que c'est une fonction « récursive ouverte ».

La fonction `memo_rec` ressemble beaucoup à la fonction `memo` de la question 11. Toutefois, au lieu d'appeler simplement `f x`, la fonction locale `memo_f` appelle `f memo_f x`.

Lorsqu'on applique enfin `memo_rec` à `c`, on obtient un résultat équivalent à la première solution présentée plus haut. En effet, dans ce cas, le paramètre `f` de `memo_rec` n'est autre que la fonction `c`. Si dans `memo_rec` on remplace `f` par `c` et si on expanse la définition de `c`, on retrouve la première solution présentée plus haut. □