Algorithmique & Programmation (INF431)
Contrôle classant CC2
CORRIGÉ

25 juin 2014

Les parties I, II et III sont indépendantes les unes des autres. Elles peuvent être traitées dans l'ordre de votre choix. Elles peuvent être traitées sur la même feuille. Au sein de chaque partie, il est souvent possible de répondre à une question même si l'on n'a pas su traiter celles qui précèdent.

Première partie

Sous-tableau maximum

Dans cette partie, on demande d'écrire **du pseudo-code** aussi clair et précis que possible.

On suppose donné un tableau a, de longueur n, dont les éléments $a[0], a[1], \ldots, a[n-1]$ sont des entiers relatifs (c'est-à-dire des éléments de $\mathbb{Z}$). On suppose $n \geq 0$.

Le **problème du sous-tableau maximum** (ou « *maximum segment sum* ») consiste à trouver, au sein du tableau a, une suite d'éléments contigus dont la somme est maximale. Formellement, on souhaite déterminer la valeur maximale de la somme $a[i] + a[i+1] + \ldots + a[j-1]$ lorsque les indices i et j varient en respectant la contrainte $0 \leq i \leq j \leq n$. (Cette somme, qui peut s'écrire également $\sum_{i \leq k < j} a[k]$, est nulle lorsque $i = j$.) On ne demande pas de calculer les indices i et j pour lesquels cette valeur maximale est atteinte.

Question 1 Donnez le pseudo-code d'une fonction MaxSegSum qui, étant donnés le tableau a et sa longueur n, renvoie un la valeur maximale d'une somme de la forme $a[i]+a[i+1]+\ldots+a[j-1]$, où $0 \leq i \leq j \leq n$. Justifiez brièvement de la correction de votre algorithme. Quelle est sa complexité asymptotique en temps et en espace ? (On ne compte pas l'espace occupé par le tableau a.) (Tout algorithme correct est accepté, mais un algorithme asymptotiquement plus efficace recevra une meilleure note.) ◇

Solution. Si a est un tableau de longueur supérieure ou égale à n, notons $maxsegsum(a, n)$ la somme optimale recherchée, c'est-à-dire la valeur maximale de la somme $a[i] + a[i+1] + \ldots + a[j-1]$ sous la contrainte $0 \leq i \leq j \leq n$.

Il semble naturel de rechercher des équations qui caractérisent $maxsegsum(a, n)$. Tout d'abord, bien sûr, on a $maxsegsum(a, 0) = 0$, car la somme d'un segment vide est nulle. Considérons maintenant le cas où n est non nul. La somme optimale $maxsegsum(a, n)$ est alors réalisée :

- soit par i et j égaux à n, auquel cas on a $maxsegsum(a, n) = 0$;
- soit par i et j inférieurs à n, auquel cas on a $maxsegsum(a, n) = maxsegsum(a, n-1)$;
- soit par i inférieur à n et j égal à n, auquel cas... ?

Dans ce dernier cas, **on n'a pas** $maxsegsum(a, n) = maxsegsum(a, n-1) + a[n-1]$. C'est faux ; cela reviendrait à calculer la somme d'éléments qui ne sont pas nécessairement consécutifs. L'équation correcte est $maxsegsum(a, n) = maxsufsum(a, n-1) + a[n-1]$, où on note $maxsufsum(a, n)$ (pour « *maximum suffix sum* ») la valeur maximale de la somme $a[i] + a[i+1] + \ldots + a[j-1]$ sous la contrainte $0 \leq i \leq j = n$.

Les équations qui caractérisent la fonction *maxsegsum* sont donc :

$$\begin{aligned} &maxsegsum(a, 0) = 0 \\ &maxsegsum(a, n) = \max(0, maxsegsum(a, n-1), maxsufsum(a, n-1) + a[n-1]) \quad \text{si } n > 0 \end{aligned}$$

Il est clair que la fonction *maxsufsum* est elle-même caractérisée par les équations suivantes :

$$\begin{aligned} &maxsufsum(a, 0) = 0 \\ &maxsufsum(a, n) = \max(0, maxsufsum(a, n-1) + a[n-1]) \quad \text{si } n > 0 \end{aligned}$$

On constate que la seconde parmi ces quatre équations peut être simplifiée :

$$maxsegsum(a, n) = \max(maxsegsum(a, n-1), maxsufsum(a, n))$$

Pour implémenter MaxSegSum, on écrit une boucle qui calcule successivement les valeurs de $maxsufsum(a, i)$ et $maxsegsum(a, i)$ pour i variant de 0 à n.

```
fonction MaxSegSum(a, n)
  maxsufsum ← 0
  maxsegsum ← 0
  pour i ← 1 à n faire
      maxsufsum ← max(0, maxsufsum + a[i − 1])
      maxsegsum ← max(maxsegsum, maxsufsum)
  renvoyer maxsegsum
```

La complexité en temps de cet algorithme est $O(n)$, ce qui est optimal, puisqu'il est impossible de calculer la somme optimale sans consulter chacun des éléments du tableau. Sa complexité en espace est $O(1)$. Elle est également optimale.

Notes du correcteur. Au lieu d'une boucle, on aurait pu écrire **une** fonction récursive qui calcule simultanément $maxsufsum(a, n)$ et $maxsegsum(a, n)$. Il fallait éviter le piège qui consisterait à transcrire littéralement les quatre équations ci-dessus en **deux** fonctions récursives $maxsufsum(a, n)$ et $maxsegsum(a, n)$, car la complexité en temps de la fonction $maxsegsum(a, n)$ ainsi obtenue serait $O(n^2)$. □

Deuxième partie

Tâches concurrentes

Dans cette partie, on demande, dans la mesure du possible, d'écrire **du code Java correct**.

On a parfois souligné en cours le fait que la création d'un « processus léger » peut être coûteuse en temps (elle demande typiquement l'exécution de plusieurs milliers d'instructions) et en mémoire (elle demande, par exemple, l'allocation d'une nouvelle pile).

Donc, lorsqu'un algorithme exige l'exécution (potentiellement parallèle) d'un grand nombre de petites tâches, il n'est pas raisonnable de créer un nouveau processus léger (ou thread) pour exécuter chaque tâche. Au lieu de cela, on souhaite confier l'exécution de l'ensemble des tâches (quel que soit leur nombre) à un nombre fixe de threads, parfois appelés « travailleurs ».

```
public interface Runnable {
  void run ();
}
```

FIGURE 1 – L'interface `Runnable`

```
public class Thread {
  public Thread (Runnable r);
  public void start ();
  // the other methods are omitted
}
```

FIGURE 2 – La classe `Thread`

```
public class AtomicInteger {
  public AtomicInteger ();
  public int get ();
  public int decrementAndGet ();
  public int incrementAndGet ();
  // the other methods are omitted
}
```

FIGURE 3 – La classe `AtomicInteger`

```
public class ReentrantLock {
  public void lock ();
  public void unlock ();
  // the other methods are omitted
}
```

FIGURE 4 – La classe `ReentrantLock`

```
public class LinkedBlockingQueue<E> {
  // This constructor creates a queue of unbounded capacity.
  public LinkedBlockingQueue ();
  // This constructor creates a queue of bounded capacity.
  public LinkedBlockingQueue (int capacity);
  public void put (E e);
  public E take ();
  // the other methods are omitted
}
```

FIGURE 5 – La classe `LinkedBlockingQueue`

Dans cette partie, nous étudions comment un tel mécanisme peut être implémenté et utilisé.

Les figures 1 à 5 rappellent différentes interfaces et classes, fournies par la librairie Java, que vous pourrez utiliser librement. On rappelle que : `Runnable` et `Thread` permettent de créer un nouveau thread (figures 1 et 2) ; `AtomicInteger` fournit un compteur partagé (figure 3) ; `ReentrantLock` fournit un verrou (figure 4) ; et `LinkedBlockingQueue` fournit une file d'attente partagée, dont la méthode `take` est bloquante, et dont la capacité peut être **bornée ou non bornée**, selon le constructeur utilisé (figure 5).

Sans nous inquiéter pour le moment de la façon dont les tâches sont construites, nous commençons par étudier la façon dont elles sont exécutées. Nous supposons qu'une tâche est représentée par un objet de type `Runnable`, donc un objet doté d'une méthode `run`, qui indique quel est le travail à exécuter. Nous souhaitons construire une classe `Manager` qui contient le code d'exécution des tâches. Cette classe doit être dotée d'un constructeur et d'une méthode `submit`, comme suit :

```
class Manager {
  Manager (int workers) { ... }
  void submit (Runnable task) { ... }
}
```

Lorsqu'on crée un nouvel objet `manager`, à l'aide de l'instruction

```
Manager manager = new Manager (workers);
```

on souhaite que les threads travailleurs soit créés et lancés immédiatement. Leur nombre est donné par l'entier `workers`. On ne s'inquiète pas de la terminaison des threads travailleurs ; ils peuvent exister éternellement.

La méthode `submit` doit ensuite permettre de soumettre une tâche : si `task` est une tâche de type `Runnable`, alors l'instruction

```
manager.submit(task);
```

doit avoir pour effet de soumettre cette tâche aux threads travailleurs : on souhaite que, dès que possible, un travailleur s'en empare et l'exécute. Nous dirons qu'une tâche est **active** dès qu'elle a été ainsi soumise. Il n'est pas nécessaire que les tâches actives soient exécutées dans l'ordre où elles ont été soumises. La méthode `submit` **ne doit pas être bloquante :** elle doit terminer immédiatement. On souligne que l'objet `manager` sera partagé et que plusieurs threads pourront appeler `submit` au même moment.

Question 2 Implémentez la classe `Manager`. Indiquez quels sont ses champs, et donnez le code du constructeur et de la méthode `submit`. ◇

Solution. La solution est donnée par la figure 6. Une file d'attente de classe `LinkedBlockingQueue` est utilisée pour stocker les tâches actives qui n'ont pas encore été saisies par un thread travailleur. Cette file doit être de capacité non bornée. Si sa capacité était bornée, alors l'appel à `put` (ligne 25) pourrait être bloquant ; or on a demandé que `submit` ne soit pas bloquante. Comme l'ordre d'exécution des tâches actives n'a pas d'importance, il n'est pas essentiel d'employer une file FIFO ; on aurait pu, par exemple, employer une pile LIFO.

Le constructeur initialise la file d'attente, puis lance les threads travailleurs, au nombre de `workers`. Tous ont le même code, dont le cœur tient en deux lignes : extraire une tâche de la file d'attente et l'exécuter (ligne 17), puis recommencer, et ce, indéfiniment (ligne 16). La méthode `take()` est bloquante, donc si aucune tâche n'est disponible pour le moment, le travailleur attend.

La méthode `submit` ajoute simplement la tâche `task` à la file d'attente. À un certain moment, dès maintenant ou un peu plus tard, un travailleur la saisira et l'exécutera.

```
1  class Manager {
2
3    // A waiting queue of tasks that are ready to be run.
4    private final BlockingQueue<Runnable> activeTasks;
5
6    // The constructor initializes the queue and spawns the worker threads.
7    Manager (int workers) {
8      // Initialize a queue of unbounded capacity.
9      activeTasks = new LinkedBlockingQueue<Runnable> ();
10     // Create the worker threads.
11     for (int i = 0; i < workers; i++) {
12       new Thread (new Runnable () {
13         public void run () {
14           // Each worker repeatedly takes a task out of the queue
15           // and runs it. If the queue is empty, it waits.
16           while (true)
17             activeTasks.take().run();
18         }
19       }).start();
20     }
21   }
22
23   // This submits a task, i.e., adds it to the queue.
24   void submit (Runnable task) {
25     activeTasks.put(task);
26   }
27
28 }
```

FIGURE 6 – La classe Manager

Notes du correcteur. La file d'attente est conçue pour être utilisée par plusieurs threads simultanément ; il n'est donc pas nécessaire de la protéger par un verrou. Il n'est pas non plus utile de maintenir un compteur du nombre d'éléments de la file.

Une erreur fréquemment rencontrée consiste à tester si la file d'attente est vide avant d'appeler `take`. C'est inutile, puisque la méthode `take` est bloquante : elle attend que la file soit non vide pour en extraire un élément. De plus, c'est néfaste : cela donne une attente active, c'est-à-dire une boucle qui interroge sans cesse la file. Certes, on peut rendre cette attente active moins coûteuse à l'aide d'un appel à `Thread.sleep`, mais on perd alors en efficacité, puisqu'on risque alors d'attendre trop longtemps. □

Voici un exemple (simpliste) d'utilisation.

Question 3 On souhaite décoder une liste de messages. Un message est représenté par un objet de type `Message`, où l'interface `Message` est définie comme suit :

```
interface Message {
  void decode ();
}
```

Écrivez une méthode `void decodeMessages (Manager manager, List<Message> messages)` qui lance le décodage de chacun des messages de la liste `messages`, de façon parallèle, c'est-à-dire en construisant une tâche par message et en soumettant ces tâches au `manager` fourni. La méthode `decodeMessages` pourra terminer immédiatement ; on ne s'inquiète pas, pour le moment, d'attendre que le décodage de tous les messages soit terminé. ◇

Solution. La solution est la suivante :

```
1  void decodeMessages (Manager manager, List<Message> messages)
2  {
3    for (final Message message : messages)
4      manager.submit (new Runnable () {
5        public void run () {
6          message.decode();
7        }
8      });
9  }
```

On parcourt la liste des messages, et pour chaque message `message`, on construit une tâche, c'est-à-dire un objet qui implémente l'interface `Runnable` et dont la méthode `run` demande le décodage de `message`. On soumet immédiatement cette tâche au `manager`. Les messages sont donc décodés en parallèle, dans la limite du nombre de travailleurs qui ont été créés.

Le mot-clef **`final`** (ligne 3) est en principe nécessaire, car Java n'autorise une classe anonyme à mentionner une variable en provenance de l'extérieur que si cette variable est non modifiable. Cependant, ce détail n'a pas été souligné en cours et n'était pas exigé.

Notes du correcteur. Cette question a le plus souvent été bien traitée. Au lieu d'employer une boucle **`for`**, comme ci-dessus, on pouvait également utiliser explicitement un itérateur :

```
Iterator<Message> it = messages.iterator();
while (it.hasNext()) {
  final Message message = it.next();
  manager.submit (new Runnable () {
    public void run () {
      message.decode();
    }
  });
}
```

Une erreur fréquemment rencontrée dans ce cas consiste à accéder à la liste `messages`, pour en retirer le prochain élément, **depuis la méthode** `run`. Le code suivant, où l'appel à `it.next()` a été déplacé dans la méthode `run`, est **incorrect** :

```
final Iterator<Message> it = messages.iterator();
while (it.hasNext()) {
  manager.submit (new Runnable () {
    public void run () {
      it.next().decode();
    }
  });
}
```

Ce code présente une race condition, et n'a pas vraiment de sens. Les tâches sont exécutées en parallèle, et risquent d'appeler `it.next()` au même moment. Or, l'objet `it` n'est pas protégé par un verrou. □

On souhaite maintenant mettre en place un mécanisme de gestion des dépendances entre tâches. Nous dirons que la tâche B **dépend** de la tâche A si l'exécution de B ne doit pas commencer avant que l'exécution de A soit terminée.

Les tâches et leurs dépendances forment ainsi un graphe orienté, dont les sommets sont les tâches, et où il existe une arête de A vers B si B dépend de A.

Nous maintiendrons **l'invariant** que les tâches actives sont exactement celles dont le degré entrant est nul. Cela signifie qu'une tâche peut et doit être soumise aux threads travailleurs dès qu'elle ne dépend d'aucune autre tâche.

On impose que le degré sortant de chaque tâche soit 0 ou 1. Une tâche A a donc au plus un successeur B. Ce cadre restreint est suffisant ici.

Le graphe n'est pas figé, mais évolue au cours du temps. Deux types de modifications sont possibles :

(M1) À tout moment, il est permis de créer une nouvelle tâche A et (optionnellement) une arête de A vers B, où B est une tâche pré-existante et inactive.

(M2) Lorsqu'une tâche active A termine son exécution, l'arête issue de A (s'il en existe une) doit disparaître.

On représente ce graphe en mémoire à l'aide d'objets de classe `Task`. La structure générale de cette classe est la suivante :

```
abstract class Task implements Runnable {
  public abstract void run ();
  Task (Manager manager) { ... }
  void setSuccessor (Task successor) { ... }
  void finished () { ... }
}
```

La classe `Task` implémente l'interface `Runnable`. Sa méthode `run` est abstraite ; elle sera implémentée dans une sous-classe de `Task`.

Le constructeur crée une nouvelle tâche A. Son argument `manager` est celui auquel cette tâche sera ultimement soumise. La nouvelle tâche n'a initialement aucune arête entrante ni sortante.

Si A est une tâche nouvellement créée (donc sans arête sortante) et si B est une tâche pré-existante et inactive, alors l'appel `A.setSuccessor(B)` crée une arête de A vers B. L'argument de `setSuccessor` peut être `null`, auquel cas il n'y a rien à faire.

Le constructeur et la méthode `setSuccessor` réalisent la modification M1 décrite plus haut.

Enfin, si A est une tâche active et qui a terminé son exécution, l'appel `A.finished()` supprime l'arête de A vers son successeur B – s'il en existe une – **et soumet la tâche B au `manager`, si celle-ci n'a plus de prédécesseurs**.

La méthode `finished` réalise la modification M2 décrite plus haut.

On souligne que, si B est une tâche existante et inactive, plusieurs threads pourront être amenés au même moment à créer une nouvelle arête de A_1 vers B, créer une nouvelle arête de A_2 vers B, supprimer une arête existante de A_3 vers B, etc. Cela ne doit pas donner lieu à une race condition : à vous d'utiliser les outils de synchronisation appropriés.

Question 4 Implémentez la classe `Task`. Indiquez d'abord quels sont ses champs. Ces champs pourront servir à stocker, entre autres, le nombre et/ou l'identité des prédécesseurs et/ou des successeurs de cette tâche dans le graphe. Donnez ensuite le code du constructeur, de la méthode `setSuccessor`, et de la méthode `finished`. ◇

Solution. Clarifions d'abord un point. Le sujet affirme : « Nous maintiendrons l'invariant que les tâches actives sont exactement celles dont le degré entrant est nul ». Par souci de concision, le sujet n'en dit pas plus. Il faut comprendre, toutefois, **quand** cet invariant doit être imposé. Il ne faut sûrement pas qu'il soit imposé automatiquement par le constructeur de la classe `Task`, car une nouvelle tâche n'a aucun prédécesseur, et il en découlerait que toute tâche nouvellement créée est immédiatement et automatiquement soumise pour être exécutée. Cela irait à l'encontre de l'idée que certaines tâches dépendent d'autres tâches. Il faut donc laisser l'utilisateur libre de construire de nouvelles tâches puis d'effectuer des appels à `setSuccessor` et/ou à `submit`, et **supposer** que l'utilisateur maintient lui-même l'invariant : il doit faire en sorte que toute tâche nouvellement créée reçoive au moins un prédécesseur **ou bien** soit soumise pour exécution.

Clarifions un second point. Dans les trois fragments de code que nous devons écrire (à savoir le constructeur, la méthode `setSuccessor`, et la méthode `finished`), quels objets de type `Task` sont peut-être partagés, et lesquels sont certainement non partagés ? Autrement dit, quand y a-t-il ou non un risque d'interférence entre plusieurs threads ?

1. Lorsque le constructeur de la classe `Task` est exécuté, l'objet `this` est nouvellement créé, et n'est donc pas encore partagé : seul le thread courant y a accès. Dans le code du constructeur, il est donc inutile de prendre un verrou pour éviter une interférence.
2. Lors d'un appel `A.setSuccessor(B)`, on peut supposer que l'objet `A` vient d'être créé (c'est ce que spécifie le sujet) et par conséquent n'est pas encore partagé. Il est donc inutile de prendre un verrou avant d'accéder aux champs de l'objet `A`. Au contraire, il faut supposer (c'est que souligne le sujet) que l'objet `B` est peut-être déjà partagé. Il faut donc utiliser un verrou (ou un autre mécanisme de synchronisation) lorsque l'on accède aux champs de l'objet `B`.
3. Lors d'un appel `A.finished()`, d'autres threads peuvent-ils accéder à l'objet `A` au même moment ? La réponse est non, car :
   — une tâche n'est exécutée qu'une fois, donc deux appels à `A.finished()` ne peuvent pas avoir lieu en même temps ;
   — une tâche active n'a pas de prédécesseurs (et n'en recevra plus), donc si un appel à `A.finished()` est en cours, alors aucun thread ne peut tenter d'ajouter ou de supprimer un prédécesseur de l'objet `A`.

   Lors d'un appel `A.finished()`, l'objet `A` peut avoir un successeur `B` qui, lui, peut être partagé (c'est que souligne le sujet). Il faut donc à nouveau utiliser un verrou (ou un autre mécanisme de synchronisation) lorsque l'on accède aux champs de l'objet `B`.

Ces considérations relativement subtiles rendent la question assez difficile. On a souvent observé des synchronisations superflues (lors des accès à `A`) et des synchronisations manquantes (lors des accès à `B`).

Une solution est donnée par la figure 7. Les champs sont les suivants. Le champ `manager` stocke simplement l'argument `manager` fourni au constructeur ; il est utilisé par la méthode

```
1  abstract class Task implements Runnable {
2
3    // The code to be executed by this task.
4    public abstract void run ();
5
6    // The manager that will run this task when ready.
7    protected final Manager manager;
8    // This task's current in-degree.
9    // This is the number of tasks that this task still depends upon.
10   private final AtomicInteger indegree;
11   // This task's (unique) successor.
12   // May be null if there is none.
13   protected Task successor;
14
15   // Create a new task with no predecessor or successor.
16   Task (Manager manager) {
17     this.manager = manager;
18     this.indegree = new AtomicInteger ();
19     this.successor = null;
20   }
21
22   // Install this task's outgoing edge.
23   void setSuccessor (Task successor)
24   {
25     assert (this.successor == null);
26     this.successor = successor;
27     if (successor != null)
28       successor.indegree.incrementAndGet();
29   }
30
31   // Remove this task's outgoing edge, if there is one.
32   void finished ()
33   {
34     // If there is an outgoing edge...
35     if (successor != null) {
36       // Decrement our successor's indegree, and
37       // if it reaches zero, ...
38       if (successor.indegree.decrementAndGet() == 0)
39         // Submit the successor task for execution.
40         manager.submit(successor);
41       // The following is not really necessary.
42       successor = null;
43     }
44   }
45
46 }
```

FIGURE 7 – La classe abstraite Task

`finished`. Comme il n'est jamais modifié après l'initialisation, on le déclare `final`. Le champ `indegree` stocke le degré entrant (c'est-à-dire le nombre de prédécesseurs) de cette tâche. Ce nombre peut varier au cours du temps, et comme on l'a souligné plus haut, plusieurs threads peuvent tenter au même moment de le modifier. On choisit ici de le stocker à l'aide d'un objet de classe `AtomicInteger`, qui représente un compteur entier modifiable et partageable. Enfin, le champ `successor` stocke l'unique successeur de cette tâche dans le graphe. Par convention, on choisit d'employer la valeur `null` pour indiquer l'absence de successeur.

Le constructeur initialise ces trois champs de façon triviale. La nouvelle tâche n'a ni arête entrante ni arête sortante.

La méthode `setSuccessor` installe une arête sortante. L'assertion ligne 25 sert de documentation, mais n'est pas nécessaire. On met à jour le champ `successor` (ligne 26) et on n'oublie pas d'incrémenter le degré entrant du sommet `successor` (ligne 28).

Dans le constructeur et dans `setSuccessor`, comme on l'a souligné plus haut, on peut supposer que l'objet `this` n'est pas encore partagé. On n'a donc pas besoin de prendre un verrou avant d'accéder à ses champs.

La méthode `finished` supprime l'arête sortante, s'il en existe une. Les deux points essentiels consistent à décrémenter le degré entrant de la tâche `successor`, et, si ce degré tombe à zéro, à soumettre cette tâche au `manager`, pour exécution dès que possible. (On suppose ici que les tâches `this` et `this.successor` ont le même objet `manager`, donc peu importe lequel des deux on utilise.)

Notes du correcteur.

Il est inutile de maintenir la liste des prédécesseurs ; maintenir leur nombre suffit.

Au lieu d'utiliser un objet de type `AtomicInteger`, on peut utiliser un simple champ de type `int`. Dans ce cas, il faut le protéger à l'aide d'un verrou. Une approche possible consiste à allouer un verrou distinct pour chaque objet `t` de type `Task`, et à le stocker dans un champ `t.lock`. Dans ce cas, il faut faire attention à verrouiller le bon verrou : lorsqu'on souhaite met à jour le champ `successor.indegree`, il faut prendre le verrou `successor.lock`, et **non pas** le verrou `this.lock`. Une autre approche, plus simple mais qui autorise un peu moins de parallélisme, consiste à allouer un seul verrou pour toutes les tâches, et à le stocker par exemple dans l'objet `manager`, dans un champ `manager.lock`. □

Voici un exemple (simpliste) d'utilisation.

Question 5 Comme lors de la question 3, on souhaite décoder une liste de messages, représentés par des objets de type `Message`. Cette fois, on souhaite de plus attendre que tous les messages soient décodés et afficher alors sur la sortie standard le message « `Done!` ». Écrivez une méthode `void decodeMessagesAndPrint (Manager manager, List<Message> messages)` qui, en créant et en soumettant des tâches appropriées, provoque le décodage en parallèle des messages de la liste `messages` puis, une fois tous les messages décodés, l'affichage du message « `Done!` ». ◇

Solution. Comme lors de la question 3, il faut construire une tâche par message, chargée du décodage. De plus, il faut une tâche supplémentaire, chargée de l'affichage du message, et qui doit dépendre de toutes les tâches de décodage. En première approximation, on aimerait écrire cela ainsi :

```
1     void decodeMessagesAndPrint (Manager manager, List<Message> messages)
2     {
3       // Create the final task.
4       Task printTask = new Task (manager) {
5         public void run () {
6           System.out.println("Done!");
7         }
8       };
```

```
9         // Create each message decoding task.
10        for (final Message message : messages) {
11          Task decodeTask = new Task (manager) {
12            public void run () {
13              // Decode the message.
14              message.decode();
15              // And signal that we are done.
16              finished();
17            }
18          };
19          // Install a dependency of printTask on this task.
20          decodeTask.setSuccessor(printTask);
21          // Submit this task immediately.
22          manager.submit(decodeTask);
23        }
24      }
```

On crée d'abord la tâche `printTask`. Sa méthode `run` contient l'instruction d'affichage. (On peut ajouter ou non un appel à `finished`.) Il **ne faut pas** appeler `manager.submit(printTask)`, car cela provoquerait l'exécution prématurée de cette tâche.

On crée ensuite chacune des tâches `decodeTask`. Comme précédemment, la méthode `run` contient le code « utile » (ligne 14). De plus, une fois le décodage de ce message terminé, il faut supprimer l'arête sortante en appelant `finished` (ligne 16). Ainsi, le degré entrant de `printTask` sera décrémenté, et lorsqu'il atteindra zéro, la tâche `printTask` sera activée.

On crée une arête de chaque tâche `decodeTask` vers la tâche `printTask` (ligne 20). Enfin, la tâche `decodeTask` n'ayant pas de prédécesseurs, il semble qu'on peut (et qu'on doit) l'activer immédiatement (ligne 22) afin de maintenir l'invariant.

Notes du correcteur.

Le code ci-dessus est en fait **incorrect**. Cette subtilité avait échappé à l'auteur du sujet (malgré un test !). Les élèves qui ont proposé la solution ci-dessus ont reçu tous les points prévus, mais ceux qui ont évité et/ou signalé le problème ont reçu des points supplémentaires. Le problème est que, comme on soumet immédiatement les tâches `decodeTask`, on court le risque que celles-ci soient exécutées immédiatement et que le degré entrant de la tâche `printTask` tombe à zéro. La tâche `printTask` est alors activée, bien qu'on n'ait pas terminé de construire les tâches `decodeTask`.

Pour clarifier le problème et sa solution, introduisons une notion de tâche **activable**. Une tâche est activable si elle risque à tout moment de devenir active. En d'autres termes, une tâche est activable : (1) si elle est déjà active ; ou bien (2) si tous ses prédécesseurs sont activables. Le sujet affirme : « on ne doit pas appeler `A.setSuccessor(B)` si la tâche `B` est active. » On comprend alors que, puisqu'une tâche activable peut devenir active à tout moment, pour respecter cette règle, il faut en fait respecter une règle plus forte : « on ne doit pas appeler `A.setSuccessor(B)` si la tâche `B` est activable. »

Dans le code ci-dessus, l'appel `manager.submit(decodeTask)` rend les tâches `decodeTask` et `printTask` toutes deux activables, ce qui fait que, lors de l'itération suivante de la boucle, l'appel `decodeTask.setSuccessor(printTask)` viole la règle ci-dessus.

Pour corriger le problème, il faut **d'abord** créer les n tâches `decodeTask` et installer les n arêtes vers la tâche `printTask`, **puis** soumettre les n tâches `decodeTask`. On écrit donc :

```
1       void decodeMessagesAndPrint (Manager manager, List<Message> messages)
2       {
3         // Create the final task.
4         Task printTask = new Task (manager) {
5           public void run () {
```

```
6           System.out.println("Done!");
7         }
8       };
9       // Create each message decoding task.
10      List<Task> decodeTasks = new ArrayList<Task> ();
11      for (final Message message : messages) {
12        Task decodeTask = new Task (manager) {
13          public void run () {
14            // Decode the message.
15            message.decode();
16            // And signal that we are done.
17            finished();
18          }
19        };
20        // Install a dependency of printTask on this task.
21        decodeTask.setSuccessor(printTask);
22        // Add this task to the list.
23        decodeTasks.add(decodeTask);
24      }
25      // Now submit all of the decoding tasks.
26      for (Task decodeTask : decodeTasks)
27        manager.submit(decodeTask);
28    }
```

Certains élèves ont ignoré le mécanisme de gestion des dépendances mis en place dans le sujet, et ont utilisé un compteur (atomique) du nombre de messages décodés et une attente active. Ce n'était pas l'esprit du sujet.

On ne pouvait pas utiliser la méthode `Thread.join`, puisque les threads travailleurs ne terminent jamais. □

Voici un autre exemple d'utilisation, où chaque tâche doit transmettre un résultat entier à son successeur. On considère le code suivant, qui calcule (très inefficacement, mais là n'est pas la question) le n-ième élément de la suite de Fibonacci :

```
int fib (int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fib(n - 1) + fib(n - 2);
}
```

On souhaite adapter ce code de façon à ce que les appels `fib(n - 1)` et `fib(n - 2)` soient exécutés en parallèle, ou plus précisément, au sein de deux tâches indépendantes.

Pour cela, on écrit une classe `FibTask`, dont la structure générale est la suivante :

```
class FibTask extends Task  {
  final int n;
  int result;
  FibTask (Manager manager, int n) { super(manager); this.n = n; }
  public void run () { ... }
}
```

Le champ `n` contient l'entier n pour lequel on souhaite calculer `fib(n)`. Le champ `result`, qui n'est pas initialisé par le constructeur, est destiné à contenir ultimement le résultat de ce calcul. Le constructeur est trivial.

Question 6 Implémentez la méthode `run` de la classe `FibTask`. ◇

```
1  class FibTask extends Task  {
2
3    // The value of n for which we are supposed to compute fib(n).
4    final int n;
5    // The result that we have computed (if this task has run).
6    int result;
7
8    FibTask (Manager manager, int n)
9    {
10     super(manager);
11     this.n = n;
12   }
13
14   public void run ()
15   {
16     if (n == 0 || n == 1)
17       // Write our result into the result field.
18       result = n;
19     else {
20       // Create two independent tasks.
21       final FibTask task1 = new FibTask (manager, n - 1);
22       final FibTask task2 = new FibTask (manager, n - 2);
23       // Create a continuation task, which waits on the two tasks
24       // above, and adds up their results.
25       Task join = new Task (manager) {
26         public void run () {
27           result = task1.result + task2.result;
28           finished();
29         }
30       };
31       // Install edges from task1 and task2 to join.
32       task1.setSuccessor(join);
33       task2.setSuccessor(join);
34       // Install an edge from join to our own successor (tricky!).
35       join.setSuccessor(this.successor);
36       // Submit task1 and task2, which have no predecessors.
37       manager.submit(task1);
38       manager.submit(task2);
39     }
40     // We are done.
41     finished();
42   }
43
44 }
```

FIGURE 8 – La classe FibTask

Solution. Une solution est donnée par la figure 8.

Le cas de base est simple. On initialise le champ `result`, et on a terminé.

Le cas général est plus complexe. Il faut créer trois nouvelles tâches, munies d'arêtes appropriées. Les tâches `task1` et `task2` sont chargées de calculer `fib(n - 1)` et `fib(n - 2)`, respectivement ; ce sont des instances de la classe `FibTask`. La tâche `join` est chargée d'additionner les résultats des deux tâches précédentes, une fois qu'elles ont terminé. Sa méthode `run` consulte donc `task1.result` et `task2.result`. (Ceci ne provoque pas de race condition, car ce code n'est exécuté qu'après la terminaison des tâches `task1` et `task2`.) Après avoir créé ces objets, on installe les arêtes appropriées, sans oublier l'arête de `join` vers `this.successor`. C'est subtil : notre successeur, qui attendait que nous ayons fini, doit maintenant attendre que la tâche `join` ait fini. Les tâches `task1` et `task2` n'ont pas de prédécesseur, donc peuvent être activées immédiatement. Ces appels à `submit` doivent avoir lieu après que toutes les arêtes de dépendances ont été installées.

Dans tous les cas, chaque tâche appelle `finished()` une fois et une seule, et en dernier.

Notes du correcteur.

La question était assez difficile et ouverte. De fait, on a observé peu de solutions correctes, et parmi celles-ci, une grande variété d'idées.

En théorie, il ne faut pas appeler `task1.setSuccessor(this)` et `task2.setSuccessor(this)`. Ces appels n'ont pas de sens, puisque la tâche `this` a déjà commencé son exécution : on n'est pas censé ajouter des prédécesseurs à une tâche déjà active.

Toutefois, certains ont imaginé qu'une tâche pouvait être interruptible et ré-activable. La première exécution crée les sous-tâches `task1` et `task2` et les dépendances appropriées, puis termine sans appeler `finished`. La seconde exécution effectue le reste du travail, c'est-à-dire l'addition `result = task1.result + task2.result` et l'appel à `finished`. Dans ce cas, on n'a pas besoin de la tâche `join`.

D'autres ont imaginé une variante de la solution proposée ci-dessus, où la tâche `this` n'appelle pas `finished()` mais, au lieu de cela, délègue la responsabilité d'appeler `finished()` à la tâche `join`, qui de ce fait n'a pas besoin d'avoir un successeur. C'est une excellente idée, à condition de ne pas se tromper de destinataire – le mot-clef `this` n'a pas la même signification partout dans le code. Dans la tâche `join`, il faut appeler `FibTask.this.finished()` et non pas `this.finished()`.

Enfin, on a vu diverses solutions spécifiques de l'exemple de Fibonacci. Par exemple, on peut imaginer qu'une tâche envoie activement son résultat à son successeur dans le graphe. Cela s'appuie en général sur le fait que l'addition est commutative et associative, et cela suppose que le successeur est lui aussi un objet de classe `FibTask`. Ces solutions n'étaient pas dans l'esprit du sujet. Elles ont reçu une partie des points. □

Troisième partie

Sélection randomisée

Notations.

Pour $i, k \in \mathbb{Z}$, on note $[i..k]$ l'ensemble $\{j \in \mathbb{Z} \mid i \leq j \leq k\}$. On note $[i..k)$ l'ensemble $[i..k-1]$.

Si a est un tableau de longueur n, alors il est indicé par l'intervalle $[0..n)$. On note $\text{len}(a)$ la longueur d'un tableau a.

Soit a un tableau de longueur n, dont les éléments sont munis d'une relation d'ordre total, et sont distincts deux à deux. Soient $i \in [0..n)$ et $k \in [1..n]$. On dit que $a[i]$ **est l'élément de rang** k de a s'il existe un sous-ensemble J de $[0..n)$ tel que :

— $|J| = k - 1$,

— $a[j] < a[i]$ pour tout $j \in J$,

— $a[j] > a[i]$ pour tout $j \in [0..n) \setminus (J \cup \{i\})$.

On note que, si le tableau a est trié par ordre croissant, alors $a[k-1]$ est l'élément de rang k de a. Toutefois, le tableau a n'est pas nécessairement trié.

Énoncé du problème.

Soit $n > 0$. Le **problème de la sélection** pour la taille n est le suivant : étant donné un tableau a de longueur n, dont les éléments sont munis d'une relation d'ordre total, et sont distincts deux à deux, et étant donné un entier $k \in [1..n]$, déterminer l'unique indice $i \in [0, n)$ tel que $a[i]$ est l'élément de rang k de a.

Dans cette partie, on souhaite analyser deux algorithmes randomisés pour ce problème. Le premier, QuickSelect, est basé sur la même idée que l'algorithme de tri QuickSort. Le second, SamplingSelect, est basé sur un échantillonnage (« *sampling* »). Tous deux sont typiquement plus efficaces que le meilleur algorithme déterministe connu (« *median of medians* »), qui a été mentionné en cours.

Conventions.

Lorsqu'on mesure la complexité en temps d'un algorithme, **on ne compte que le nombre de comparaisons** (entre deux éléments de a) effectuées par l'algorithme. On suppose qu'une seule opération de comparaison entre x et y suffit à distinguer les trois cas $x < y$, $x = y$, et $x > y$.

On s'intéresse toujours à la complexité dans le pire cas vis-à-vis de l'entrée, c'est-à-dire à la complexité maximale de l'algorithme vis-à-vis de tous les tableaux a de longueur n et de tous les entiers $k \in [1..n]$. On appelle cette quantité « complexité » tout court.

Dans la suite, on utilise l'expression « **complexité dans le pire cas** » pour faire référence au **maximum** de la complexité vis-à-vis des décisions aléatoires prises par l'algorithme. On utilise l'expression « **complexité espérée** » pour faire référence à **l'espérance** de la complexité vis-à-vis des décisions aléatoires prises par l'algorithme.

On rappelle que la notation $O(n^{17})$ représente une **borne supérieure** (à une constante près). La notation $\Omega(n^{17})$ représente une **borne inférieure** (à une constante près). La notation $\Theta(n^{17})$ représente la conjonction des deux, c'est-à-dire une **estimation exacte** (à une constante près).

Analyse de QuickSelect

On considère l'algorithme récursif QuickSelect (Algorithme 1).

Question 7 Quelle est la complexité dans le pire cas de cet algorithme ? Donnez une estimation asymptotique exacte (par exemple, $\Theta(n^{17})$). Justifiez séparément et brièvement la borne supérieure (pour cet exemple, $O(n^{17})$) et la borne inférieure (pour cet exemple, $\Omega(n^{17})$). ⋄

Solution. We first have to note that, unfortunately, the algorithm as given here contains a small bug. The reason is that we tried to formulate it in a way that the algorithm returns not the k-smallest element, but its index in a. This is the more natural thing to look for, but in this case, it is harder to compute because elements of a change their position due to the partition procedure. Hence the better formulation of the algorithm would have been as follows : We only aim at returning the element of rang k in a (change line 2 accordingly). In line 9, we return QuickSelect$(c, k - (n - \text{len}(c)))$ without the faulty index shift "$+(n - \text{len}(c))$". In line 11, we return simply $a[p]$. As all changes we did concern only the return arguments, which also in the recursive

Algorithm 1: L'algorithme QuickSelect (pseudo-code)

Entrée : Un tableau a de longueur n, dont les éléments sont distincts deux à deux, et un entier $k \in [1..n]$;
Sortie : L'indice i tel que $a[i]$ est l'élément de rang k de a;
Choisir $p \in [0..n)$ aléatoirement et uniformément;
(**Partition.**) Construire deux tableaux b et c tels que b contient les éléments de a strictement inférieurs à $a[p]$ et c contient les éléments de a strictement supérieurs à $a[p]$; cette étape exige $n - 1$ comparaisons;
if $k \leq \text{len}(b)$ **then**
    return QuickSelect(b, k)
else
    if $k > n - \text{len}(c)$ **then**
        return QuickSelect$(c, k - (n - \text{len}(c))) + (n - \text{len}(c))$
    else
        return (p)

Algorithm 2: L'algorithme SamplingSelect (pseudo-code)

Entrée : Un tableau a de longueur n impaire, dont les éléments sont distincts deux à deux;
Sortie : L'élément median de a ou, rarement, "FAIL";
$N := \lceil n^{3/4} \rceil$;
for $i = 0$ **to** $N - 1$ **do**
    choisir $j \in [0..n)$ aléatoirement et uniformément;
    $s[i] := a[j]$;
$t :=$ MergeSort(s);
$\ell := \lfloor (N/2) - \sqrt{n} \rfloor - 1$;
$r := \lceil (N/2) + \sqrt{n} \rceil - 1$;
$c := (n + 1)/2$;
(**Partition.**) Construire trois tableaux L, M et R tels que L contient les $a[i]$ tels que $a[i] < t[\ell]$, M contient les $a[i]$ tels que $t[\ell] < a[i] < t[r]$, et R contient les $a[i]$ tels que $a[i] > t[r]$;
if $\text{len}(L) > c - 2$ **or** $\text{len}(R) > c - 2$ **or** $\text{len}(M) > 4N$ **then return** "FAIL";
$M' :=$ MergeSort(M);
return $M'[c - \text{len}(L) - 2]$;

calls have no influence on the further behaviour of the algorithm, the corrected algorithm and the faulty algorithm have exactly the same runtime behaviour. In such, fortunately, this mistake has no influence on the exam. Of course, we still regret this mistake and apologize.

The worst-case runtime of QuickSelect is $\Theta(n^2)$.

Upper bound : Each two elements of a are compared at most once. Or : Let W_n be the worst-case performance of QuickSelect on input arrays of size n or smaller. Then $W_1 = 0$ and $W_n \leq (n-1) + W_{n-1}$ for all $n \geq 2$. Hence an easy induction shows $W_n = O(n^2)$.

Lower bound : Assume that $k = n$. If we are unlucky, the pivot element $a[p]$ is always the smallest element of the array. In this case, we recurse with an array of length smaller by one only. In total, we call QuickSelect once with each array length between 2 and n. Since a call with an array of length i generates $i-1$ comparisons in addition to those stemming from possible recursive calls, this makes a total of $\Omega(n^2)$ comparisons. □

Pour $n > 0$, pour $k \in [1..n]$, et pour un tableau a de longueur n, on note $T(a,k)$ le nombre **espéré** de comparaisons effectuées par l'appel QuickSelect(a,k). On note $T(n,k)$ le maximum des $T(a,k)$ vis-à-vis de tous les tableaux a de longueur n possibles. On note $T(n)$ le maximum des $T(n,k)$ vis-à-vis de tous les $k \in [1..n]$.

Question 8 Donnez (et justifiez) une borne supérieure pour $T(n)$, exprimée en fonction des $T(n')$, pour $n' < n$. ◇

Solution. We have $T(1) = 0$ since for $n = 1$ no comparisons are performed. Consider a call of QuickSelect with parameters n and k. The Split procedure first uses $n-1$ comparisons. If the random $p \in [0..n)$ is such that $a[p]$ has rank j in a, then the arrays b and c are of sizes $j-1$ and $n-j$. We can pessimistically assume that we always (including when $j = k$) recurse with the array that leads to the larger complexity. The latter is bounded from above by $\max\{T(j-1), T(n-j)\}$, where we use the artificial notation $T(0) := 0$. Each rank $j \in [1..n]$ occurs with the same probability $1/n$. Consequently

$$\begin{aligned} T(n) &= \max\{T(n,k) \mid k \in [1..n]\} \\ &\leq \max\left\{ n - 1 + \sum_{j \in [1..n]} \Pr(a[p] \text{ has rank } j \text{ in } a)\ \max\{T(j-1), T(n-j)\} \,\middle|\, k \in [1..n] \right\} \\ &\leq n - 1 + (1/n) \sum_{j \in [1..n]} \max\{T(j-1), T(n-j)\}. \end{aligned}$$

Question 9 À l'aide de la borne proposée lors de la question 8, démontrez que, pour tout $n > 0$, on a $T(n) \leq 4n$. ◇

Solution. Using the inductive assumption $T(n') \leq 4n'$ for all $n' < n$, we have

$$T(n) \leq n - 1 + (4/n) \sum_{j \in [1..n]} \max\{j-1, n-j\} \leq 4n.$$

Here the last inequality stems from the following observations. For n odd, we have $\sum_{j \in [1..n]} \max\{j-1, n-j\} = 2\sum_{j=(n+1)/2}^{n-1} j + (n-1)/2 \leq (3/4)n^2$, where for the last estimate we may use that $(n-1-i) + ((n+1)/2 + i) \leq 2 \cdot (3/4)n$ and trivially $(n-1)/2 \leq (3/4)n$. For even n, we compute similarly $\sum_{j \in [1..n]} \max\{j-1, n-j\} = 2\sum_{j=n/2}^{n-1} j \leq (3/4)n^2$. □

Dans de nombreux algorithmes récursifs, on arrête la récursivité avant d'atteindre le cas de base le plus élémentaire ; au lieu de cela, on résoud les problèmes de « petite » taille à l'aide d'une méthode particulière.

Question 10 Imaginons que pour tout $n \leq 16$, vous parveniez à résoudre le problème de la sélection à l'aide de $c'n$ comparaisons, où $c' < 4$. De combien cela permettrait-il d'améliorer la borne $T(n) \leq 4n$ pour n quelconque ? Donnez une estimation exacte (par exemple, $\Theta(\sqrt{n})$) du **gain**, c'est-à-dire du nombre de comparaisons économisées. Justifiez-la brièvement. ⋄

Solution. The improvement will only be of order $\Theta(1)$. In a (recursive) run of QuickSelect, the QuickSelect function is called at most once for each problem size n. Consequently, improving the performance of QuickSelect for $n \in [1..16]$ each by an additive constant also gives in total at most an additive improvement, showing that the gain is $O(1)$ only. That there is some $\Omega(1)$ gain at all stems from the fact that with probability at least $\Omega(1)$, a run of SamplingSelect will encounter a SamplingSelect call with an array of size between 1 and 16, and there we save a constant amount of work. □

Question 11 Même question, cette fois à propos de l'algorithme QuickSort étudié en cours. Imaginons que pour tout $n \leq 16$, vous parveniez à trier un tableau de n éléments en utilisant 10% de comparaisons de moins que QuickSort. Vous modifiez donc QuickSort pour utiliser cette méthode dès que $n \leq 16$. De combien cela améliore-t-il la complexité espérée de QuickSort pour une taille n arbitraire ? Donnez une estimation exacte du gain. Justifiez-la brièvement. ⋄

Solution. $\Theta(n)$. In the recursive splitting of the array to be sorted, typically a linear number of subproblems with constant size occur. If for each a small constant improvement is obtained, we save in total $\Theta(n)$ time. □

Sélection par échantillonnage

On s'intéresse maintenant à une autre technique standard dans le domaine des algorithmes randomisés : l'échantillonnage. L'idée est de choisir aléatoirement une petite partie des données initiales, l'analyser (plus efficacement, puisqu'elle est plus petite), et utiliser le résultat de cette analyse pour résoudre le problème entier.

L'algorithme SamplingSelect (Algorithme 2) applique cette idée au problème de la sélection du médian.

On suppose, dans toute la suite, que n est impair. On cherche uniquement à trouver l'élément médian, c'est-à-dire l'élément de rang $c = (n+1)/2$. On pourra supposer n « suffisamment grand » partout où cela sera utile ou nécessaire pour expliquer le bon fonctionnement de l'algorithme.

Question 12 Quelle est la complexité dans le pire cas de SamplingSelect ? Donnez une estimation exacte (par exemple, $\Theta(n^{17})$). Justifiez séparément et brièvement la borne supérieure (pour cet exemple, $O(n^{17})$) et la borne inférieure (pour cet exemple, $\Omega(n^{17})$). ⋄

Solution. $\Theta(n)$.

$O(n)$: Comparisons occur in the two calls of MergeSort and in the Split procedure. The two MergeSort calls are on arrays of length $O(N)$, so they lead to $O(n^{3/4} \log n)$ comparisons. Hence the most costly part is the Split-Procedure, which can be implemented using $2(n-2)$ comparisons in the worst-case.

$\Omega(n)$: The Split procedure, in any case, needs at least $n/2$ comparisons, because otherwise one elements was never part of a comparison. □

On souligne le fait que l'étape de partition découpe par la pensée le tableau a en cinq portions, à savoir le tableau L, l'élément $t[\ell]$, le tableau M, l'élément $t[r]$, et le tableau R.

Question 13 À la ligne 12 de l'algorithme SamplingSelect, un échec a lieu si au moins l'une des trois conditions suivantes est satisfaite : (i) $\text{len}(L) > c - 2$, (ii) $\text{len}(R) > c - 2$, (iii) $\text{len}(M) > 4N$. Pour chacune de ces conditions, expliquez informellement et brièvement ce que cette condition signifie et dans quel but on l'a fait apparaître (a contrario, que se passerait-il si elle était omise ?).⋄

Solution. If one of (i), (ii) is satisfied, then the median we are looking for is not contained in M, but in (i) L or (ii) R. Omitting it would lead to a failure, because in line 14 we would access an array element that is out of bounds. Condition (iii) ensures that in the non-failure case, M is not too large, so the sorting does not take too long. If we omit it, the algorithm would still be correct, but sorting M could take up to $\Theta(n \log n)$ comparisons. □

Notons m l'élément médian du tableau a. On observe (sans démonstration) que la condition « $\text{len}(R) > c-2$ » est équivalente à la condition « au moins r éléments du tableau s sont inférieurs ou égaux à m ».

Notons X le nombre d'éléments du tableau s qui sont inférieurs ou égaux à m. X est une variable aléatoire. L'événement « $\text{len}(R) > c-2$ » est équivalent à l'événement « $X \geq r$ ».

Question 14 Montrez que l'espérance de X satisfait $E[X] = (1/2)N(1 + 1/n)$. ◇

Solution. Let X_i be the indicator random variable for the event that $s[i] \leq m$. Let $X = \sum_{i \in [0..N)} X_i$ be the number of $s[i]$ that are not larger than m. Then $\Pr[X_i = 1] = c/n$ and thus $E[X] = N(c/n) = (1/2)N(1 + 1/n)$ by linearity of expectation. □

Question 15 Montrez que la variance de X satisfait $\text{Var}[X] = O(n)$. ◇

Solution. By independence of the X_i, the variance satisfies $\text{Var}[X] = \sum_{i \in [0..N)} \text{Var}[X_i] = N \,\text{Var}[X_1] = \Theta(N)$. □

On rappelle l'inégalité de Tchebychev :

$$\Pr(|X - E[X]| \geq \alpha) \leq \frac{\text{Var}[X]}{\alpha^2}.$$

Question 16 À l'aide de l'inégalité de Tchebychev, déduisez des deux résultats précédents le fait que $\Pr[X \geq r] = O(n^{-1/4})$. ◇

Solution. $\Pr[X \geq r] \leq \Pr[|X - E[X]| \geq n^{1/2} - 1] \leq \text{Var}[X]/(n^{1/2} - 1)^2 = O(n^{-1/4})$. □

Ainsi, la condition « $\text{len}(R) > c-2$ » (ligne 12) n'est satisfaite qu'avec une probabilité $O(n^{-1/4})$. On peut montrer de façon analogue que les événements « $\text{len}(L) > c-2$ » et « $\text{len}(M) > 4N$ » ont lieu chacun avec probabilité $O(n^{-1/4})$. Par conséquent, SamplingSelect renvoie "FAIL" avec une probabilité $O(n^{-1/4})$ seulement.

Question 17 Montrez que l'étape de **Partition** (ligne 11) peut être implémentée de manière à utiliser au plus $(3/2)n + o(n)$ comparaisons dans le cas où $\text{len}(L) \leq c-2$ et $\text{len}(R) \leq c-2$ et $\text{len}(M) \leq 4N$. On ne demande pas nécessairement de pseudo-code ; une description informelle suffit. ◇

Solution. We implement the Split procedure as follows : For each $i \in [0..n)$, we first check if $a[i] < t[\ell]$ (equivalent to $a[i] \in L$). If this is not the case, we check if $a[i] > t[r]$ (equivalent to $a[i] \in R$). If also this is not true, then with another at most two comparisons we find out whether $a[i] \in M$. Assume that $\text{len}(L) \leq c-2$ and $\text{len}(R) \leq c-2$ and $\text{len}(M) \leq 4N$. Then there are at most $c - 2 \leq n/2$ elements of a (those that lie in L) that use exactly one comparison, there are at most $c - 2 \leq n/2$ that use exactly two comparisons, and only the $\text{len}(M) + 2 = o(n)$ remaining elements take up to four. This makes a total of $(3/2)n + o(n)$ comparisons. Note : Formally, we do not need any additional comparisons for the elements in M, since we assumed that the previous comparisons tell us which of the three results "smaller", "equal" and "greater" is true. Hence the elements of M only use 2 comparisons. This, however, does not give a result better than $(3/2)n + o(n)$. □

On suppose dans la suite que l'étape de partition est implémentée de la manière suggérée par la question 17.

Des résultats des questions 16 et 17, il découle alors immédiatement que SamplingSelect, avec probabilité $1 - O(n^{-1/4})$, utilise au plus $(3/2)n + o(n)$ comparaisons.

Question 18 Indiquez comment modifier l'algorithme SamplingSelect de sorte qu'il n'échoue jamais. On ne demande pas nécessairement de pseudo-code ; une description informelle suffit. Quelle est la complexité dans le pire cas de l'algorithme ainsi obtenu ? Quelle est sa complexité espérée ? (Donnez une borne supérieure précise jusqu'aux termes d'ordre inférieur, par exemple $19n + o(n)$.) Justifiez vos affirmations. ◇

Solution. Repeat SamplingSelect until a successful run occurs and return the result of the successful run. There is no worst-case performance guarantee, since in each iteration we could be unlucky and obtain a failure. Since the probability for a run to be successful is a nice $p = 1 - O(n^{-1/4})$, the expected number of failed runs before the final (successful) run is $o(1)$. Since each unsuccessful run uses $O(n)$ comparisons with probability one, the unsuccessful runs use an expected number of $o(n)$ comparisons. Together with the successful run using $(3/2)n + o(n)$ comparisons, we obtain a total expected performance of $(3/2)n + o(n)$. □