Algorithmique & Programmation (INF431)
Contrôle classant CC2

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.) ◇

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 ».

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`. ◇

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. ◇

```
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`

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`. ◇

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!` ». ◇

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`. ◇

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, $\Omega(n^{17})$). ◇

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 $n' < n$. ◇

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

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. ◇

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 $M'[c - \text{len}(L) - 2]$;

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. ◇

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, $\Omega(n^{17})$). ◇

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 ?).◇

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 $E[X] = (1/2)N(1 + 1/n)$. ◇

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

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 $\Pr[X \geq r] = O(n^{-1/4})$. ◇

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. ◇

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. ◇