Outils logiques et algorithmiques – Partiel – Mars 2024

Durée 2h. Le sujet comporte 3 exercices indépendants. Notes personnelles et documents de cours autorisés.
Barème approximativement proportionnel au temps nécessaire à la résolution de chaque question.

Exercice 1. Petit calcul *(10 minutes)*

Montrer que, pour tout entier $n > 0$, on a

$$\sum_{1 \leq k \leq n} \frac{1}{2^k} \quad = \quad 1 - \frac{1}{2^n}$$

Technique au choix : récurrence ou calcul astucieux.

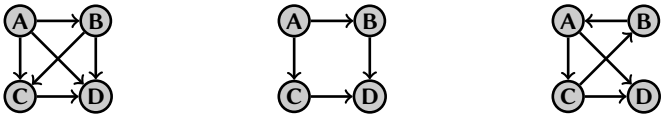
Exercice 2. Chemins hamiltoniens dans un graphe orienté acyclique *(40 minutes)*

Dans cet exercice, on considère exclusivement des graphes *orientés*. On appelle ***chemin hamiltonien*** dans un graphe G un chemin qui passe exactement une fois par chaque sommet du graphe. Dans le graphe ci-dessous à gauche, on a exactement un chemin hamiltonien : $A \longrightarrow C \longrightarrow B \longrightarrow D$. Dans le graphe ci-dessous à droite, il n'y en a pas.

Rappels : un cycle est un chemin dont le sommet de départ et le sommet d'arrivée sont égaux, et qui emprunte au moins une arête. Un graphe acyclique est un graphe dans lequel il n'y a pas de cycle. Un tri topologique d'un graphe $G = (S, A)$ est une séquence T contenant tous les sommets de S, telle que s'il existe une arête $s_1 \longrightarrow s_2$ dans A, alors s_1 apparaît avant s_2 dans T. Un graphe orienté admet au moins un tri topologique si et seulement s'il est acyclique.

A. Généralités.

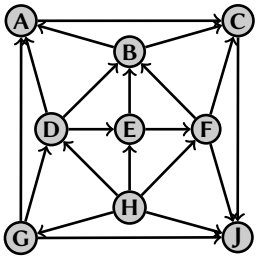
1. Pour le graphe ci-dessus à gauche, donner le degré entrant et le degré sortant de chaque sommet.
2. Pour chacun des graphes suivants, donner un chemin hamiltonien s'il en existe un.



3. La présence d'arêtes parallèles ou de boucles peut-elle influer sur l'existence d'un chemin hamiltonien ? (justifier)
4. Montrer que si un graphe G contient deux chemins hamiltoniens distincts, alors G contient nécessairement au moins un cycle.
5. Montrer que si G est un graphe *acyclique* dans lequel deux sommets ont le degré entrant 0, alors G ne peut pas contenir de chemin hamiltonien.

B. Conception d'un algorithme.

6. Supposons qu'un graphe G *acyclique* contienne un chemin hamiltonien. Montrer qu'il existe un unique tri topologique de G (il faut montrer l'existence *et* l'unicité).
7. Supposons qu'un graphe G admette un tri topologique T dans lequel au moins deux sommets consécutifs s_1 et s_2 *ne sont pas* reliés par une arête. Montrer que la séquence T' obtenue en échangeant s_1 et s_2 dans T est encore un tri topologique de G.
8. Déduire des questions précédentes qu'un graphe G *acyclique* possède un chemin hamiltonien si et seulement s'il admet un tri topologique T dans lequel chaque sommet est relié au suivant par une arête.
9. Décrire informellement un algorithme déterminant si un graphe acyclique possède un chemin hamiltonien, de complexité proportionnelle à la taille du graphe (nombre de sommets + nombre d'arêtes).
10. Appliquer l'algorithme de la question précédente au graphe suivant pour y trouver un chemin hamiltonien, s'il en existe un.



Exercice 3. Somme maximale *(70 minutes)*

On se donne un tableau de nombres entiers qui peuvent être positifs ou négatifs, et on cherche à calculer la plus grande somme d'éléments consécutifs. Dans le tableau ci-dessous

1	−4	2	−1	0	3	−2

la somme maximale est 4, obtenue en ajoutant les éléments de la séquence | 2 | −1 | 0 | 3 |. Dans cet exercice, nous allons considérer quatre algorithmes différents pour résoudre ce problème. Les quatre parties sont essentiellement indépendantes.

A. Solution directe (`maxSum1`). Une approche directe consiste à calculer les sommes des éléments dans tous les segments du tableau `t`, et à mémoriser la plus grande somme trouvée. Cette approche est mise en œuvre par l'algorithme `maxSum1` (code en bas de la page). Cette version utilise une fonction auxiliaire `sum(t, lo, hi)` qui calcule la somme des éléments de `t` entre l'indice `lo` inclus et l'indice `hi` exclu. Cette fonction `sum` est documentée ainsi :

entrées : un tableau `t` de n éléments, et deux indices `lo` et `hi`

précondition : $0 \le \texttt{lo} \le \texttt{hi} \le n$

résultat : $\texttt{r} = \sum_{\texttt{lo} \le k < \texttt{hi}} \texttt{t}[k]$

complexité : effectue `hi` − `lo` accès au tableau `t`

Questions.

1. Donner une spécification pour le problème de la somme maximale.
2. Que dire si le tableau ne contient que des nombres positifs ? Ou que des nombres négatifs ?
3. Lors de l'appel `maxSum1(t)` avec `t` un tableau de taille n, combien de fois est appelée la fonction `sum` ?
4. En déduire un ordre de grandeur du nombre d'accès au tableau pour `maxSum1`.

B. Solution directe améliorée (`maxSum2`). L'algorithme `maxSum2` conserve l'idée du précédent, mais évite les calculs redondants de la fonction auxiliaire `sum`. Pour cela, elle effectue progressivement le calcul des sommes à partir d'un indice inférieur `lo` donné.

Questions.

5. Détailler l'exécution de `maxSum2` sur le tableau | −2 | 3 | −2 | 3 |, sous la forme d'un tableau donnant les valeurs de `lo`, `k`, `sum` et `maxSum` après chaque passage par la ligne 8.
6. Combien la fonction `maxSum2` réalise-t-elle d'accès à des éléments du tableau `t`, si celui-ci a une taille n ? Donner le nombre exact, un ordre de grandeur (Θ) et un équivalent (~).
7. Donner, sous la forme d'une équation mathématique, un invariant de la boucle interne (ligne 6), caractérisant la valeur de la variable `sum`.
8. Donner, sous la forme d'une formule logique, un invariant de la boucle externe (ligne 4), caractérisant la valeur de la variable `maxSum`.
9. Donner un invariant de la boucle interne (ligne 6), caractérisant la valeur de la variable `maxSum`.

```
1  static int maxSum1(int[] t) {
2      int n = t.length;
3      int maxSum = 0;
4      for (int lo=0; lo<n; lo++) {
5          for (int hi=lo+1; hi<=n; hi++) {
6              int sum = sum(t, lo, hi);
7              if (sum > maxSum) { maxSum = sum; }
8          }
9      }
10     return maxSum;
11 }
```

```
1  static int maxSum2(int[] t) {
2      int n = t.length;
3      int maxSum = 0;
4      for (int lo=0; lo<n; lo++) {
5          int sum = 0;
6          for (int k=lo; k<n; k++) {
7              sum += t[k];
8              if (sum > maxSum) { maxSum = sum; }
9          }
10     }
11     return maxSum;
12 }
```

```
1  static int maxSum3(int[] t) {
2      int n = t.length;
3      int maxSum = 0;
4      int sum = 0;
5      for (int k=0; k<n; k++) {
6          sum += t[k];
7          if (sum > maxSum) { maxSum = sum; }
8          if (sum < 0) { sum = 0; }
9      }
10     return maxSum;
11 }
```

C. Solution maline (`maxSum3`, algorithme de Kadane). L'algorithme de Kadane (code `maxSum3` au bas de la page précédente) propose une solution plus efficace au problème de la somme maximale. On y traverse une seule fois le tableau, en maintenant deux variables : `maxSum` contient la plus grande somme observée jusque là, et `sum` la plus grande somme d'un suffixe du segment déjà traversé. Rappel : un suffixe du segment $[0, k[$ est un segment $[i, k[$.
Questions.

10. Détailler l'exécution de `maxSum3` sur le tableau | 2 | −1 | −3 | 3 | −1 | 2 | −6 | 1 |, sous la forme d'un tableau donnant les valeurs de `k`, `sum` et `maxSum` après chaque passage par la ligne 8.
11. Combien `maxSum3` réalise-t-elle d'accès au tableau pris en entrée ?
12. Donner un invariant de la boucle à propos de la valeur de la variable `sum`.
13. En supposant que, au début d'un tour de boucle, `maxSum` est la somme maximale d'un segment de `t`$[0,$`k`$[$ et `sum` est la somme maximale d'un suffixe de `t`$[0,$`k`$[$, démontrer qu'à la fin de ce tour de boucle `maxSum` est la somme maximale d'un segment de `t`$[0,$`k`$]$ et `sum` est la somme maximale d'un suffixe de `t`$[0,$`k`$]$.

D. Solution récursive (`maxSum4`, diviser pour régner). L'algorithme `maxSum4` (code en bas de cette page) calcule 4 informations sur le tableau `t` pris en entrée. Le résultat est renvoyé sous la forme d'un tableau `q` de taille 4, dont les indices correspondent aux informations suivantes :

`q[0]` : la somme maximale à l'intérieur du segment `t`$[$`lo`$,$`hi`$[$,

`q[1]` : la somme maximale parmi les préfixes du segment `t`$[$`lo`$,$`hi`$[$ (les segments qui commencent à l'indice `lo` inclus),

`q[2]` : la somme maximale parmi les suffixes du segment `t`$[$`lo`$,$`hi`$[$ (les segments qui terminent à l'indice `hi` exclu),

`q[3]` : la somme totale du segment `t`$[$`lo`$,$`hi`$[$.

L'algorithme procède récursivement, en calculant d'abord ces mêmes informations séparément pour la moitié gauche et pour la moitié droite du segment considéré.
Questions.

14. En supposant que M_1 et M_2 sont respectivement la somme maximale du segment `t`$[$`lo`$,$`mid`$[$ et celle du segment `t`$[$`mid`$,$`hi`$[$, montrer qu'il serait incorrect de considérer `max(`M_1`,` M_2`)` comme étant la somme maximale du segment `t`$[$`lo`$,$`hi`$[$.
15. En supposant que P est la somme maximale des préfixes du segment `t`$[$`lo`$,$`mid`$[$, montrer qu'il serait incorrect de considérer P comme étant la somme maximale des préfixes du segment `t`$[$`lo`$,$`hi`$[$.
16. Justifier que le résultat renvoyé par `maxSum4(t, lo, hi)` lorsque `lo = hi` est correct.
17. Expliquer pourquoi le résultat renvoyé par `maxSum4(t, lo, hi)` lorsque `lo = hi` -1 est parfois incorrect. Comment le corriger ?
18. Dans le cas où `lo < hi` -1, et en supposant que les résultats renvoyés par les appels récursifs `maxSum4(t, lo, mid)` et `maxSum4(t, mid, hi)` sont corrects, justifier que le résultat final de `maxSum4(t, lo, hi)` est correct.
19. En considérant que le coût d'un appel `maxSum4(t, lo, hi)` est borné par une constante a, à laquelle s'ajoutent les coûts des éventuels appels récursifs, donner des équations pour la complexité $C(n)$ de `maxSum4` en fonction de la taille n du segment analysé.
20. Résoudre les équations précédentes dans le cas où n est une puissance de 2, et en déduire un ordre de grandeur pour $C(n)$. Comparer avec l'algorithme de Kadane (`maxSum3`).

```
1  static int maxSum4(int[] t) {
2      return maxSum4(t, 0, t.length)[0];
3  }
4  static int[] maxSum4(int[] t, int lo, int hi) {
5      if (lo == hi)  return new int[]{0, 0, 0, 0};
6      if (lo == hi-1) return new int[]{t[lo], t[lo], t[lo], t[lo]};
7      int mid = lo + (hi-lo)/2;
8      int[] mslo = maxSum4(t, lo, mid);
9      int[] mshi = maxSum4(t, mid, hi);
10     return new int[]{
11         max(mslo[0], mshi[0], mslo[2]+mshi[1]),
12         max(mslo[1], mslo[3]+mshi[1]),
13         max(mshi[2], mshi[3]+mslo[2]),
14         mslo[3]+mshi[3]
15     };
16 }
```

Note java : l'expression

```
new int[]{a, b, c, d}
```

crée un tableau de taille 4, contenant dans l'ordre les éléments `a`, `b`, `c`, et `d`. On suppose une fonction `max` surchargée avec les signatures suivantes :

```
int max(int a, int b)
int max(int a, int b, int c)
```

Correction exercice 1.

Démonstration par récurrence sur n.

— Cas de base : $n = 1$.

$$\sum_{1 \leq k \leq 1} \frac{1}{2^k} = \frac{1}{2^1} = \frac{1}{2} = 1 - \frac{1}{2} = 1 - \frac{1}{2^1}$$

— Hérédité. Soit $n \geq 1$ tel que $\sum_{1 \leq k \leq n} \frac{1}{2^k} = 1 - \frac{1}{2^n}$. Alors

$$\sum_{1 \leq k \leq n+1} \frac{1}{2^k} = \left(\sum_{1 \leq k \leq n} \frac{1}{2^k} \right) + \frac{1}{2^{n+1}} = 1 - \frac{1}{2^n} + \frac{1}{2^{n+1}} = 1 - \frac{1}{2^{n+1}}$$

Par principe de récurrence, pour tout $n \geq 1$ on a $\sum_{1 \leq k \leq n} \frac{1}{2^k} = 1 - \frac{1}{2^n}$.

Correction exercice 2.

1.

	A	B	C	D
degré entrant	0	2	1	2
degré sortant	2	1	2	0

2. De gauche à droite :
   - $A \longrightarrow B \longrightarrow C \longrightarrow D$ (seule possibilité)
   - Pas de chemin hamiltonien, car aucun chemin ne peut passer à la fois par B et par C.
   - $C \longrightarrow B \longrightarrow A \longrightarrow D$, ou $B \longrightarrow A \longrightarrow C \longrightarrow D$
3. Non, aucune incidence. Un chemin hamiltonien ne passe jamais par une boucle (le passage par une boucle $s \longrightarrow s$ impliquerait deux visites du sommet s), et n'emprunte qu'une seule arête $s_1 \longrightarrow s_2$ entre deux sommets s_1 et s_2 donnés (sinon, on visiterait deux fois chacun de ces deux sommets). Donc s'il existe un chemin hamiltonien dans un graphe avec boucles et arêtes multiples, il en existe encore un dans le même graphe privé de ses boucles et arêtes multiples. Réciproquement, s'il existe un chemin hamiltonien dans un graphe G, ce chemin existe encore dans tout graphe G' obtenu en ajoutant des arêtes à G (que ces arêtes soient ou non des boucles ou des arêtes parallèles à des arêtes déjà présentes).
4. Supposons que le graphe $G = (S, A)$ contienne deux chemins hamiltoniens $s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n$ et $t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_n$. Par définition, chacune des ces deux séquences contient exactement une fois chaque sommet de S.
   Notons i le premier indice tel que $s_i \neq t_i$. Notons j l'indice auquel la séquence des s atteint le sommet t_i (c'est-à-dire : $s_j = t_i$). On a nécessairement $j > i$ (si $j < i$, alors $t_j = s_j$ et on a donc deux occurrences de t_i dans la séquence des t), et donc la séquence des s donne un chemin $s_i \longrightarrow \ldots \longrightarrow s_j$. On a de même un chemin $t_i \longrightarrow \ldots \longrightarrow t_k$ où $t_k = s_i$. En combinant ces deux chemins on obtient $s_i \longrightarrow \ldots \longrightarrow s_j = t_i \longrightarrow \ldots \longrightarrow t_k = s_i$, qui est un cycle.
5. Un sommet de degré entrant zéro ne peut apparaître qu'en première position de tout chemin. Un même chemin ne peut donc pas visiter deux sommets de degré entrant zéro distincts.
6. Notons $s_1 \longrightarrow \ldots \longrightarrow s_n$ le chemin hamiltonien. Montrons que la séquence $T = s_1, \ldots, s_n$ est un tri topologique de G. Soit $s_i \longrightarrow s_j$ une arête de G. Si $i > j$, alors on a un cycle $s_j \longrightarrow s_{j+1} \longrightarrow \ldots \longrightarrow s_i \longrightarrow s_j$ contredisant l'acyclicité de G. On a de même un cycle si $i = j$ (l'arête ciblée étant alors une boucle). Donc $i > j$, c'est-à-dire s_i apparaît bien avant s_j dans la séquence.
   Montrons également qu'il n'y a pas d'autre tri topologique possible. Si T' est un autre tri topologique, alors il existe un i tel que s_{i+1} apparaît avant s_i dans T'. Or il existait une arête $s_i \longrightarrow s_{i+1}$: contradiction.
7. Soit une arête $s \longrightarrow s'$ dans G. Comme la séquence T est un tri topologique de G, le sommet s y apparaît avant le sommet s'. Raisonnement par cas sur les comparaisons de s et s' avec s_1 et s_2.
   - Si ni s ni s' n'est égal à s_1 ni à s_2, alors s et s' conservent leurs positions dans T', et s y apparaît bien avant s'.
   - Si s est égal à s_2, alors s' apparaît après s_1 et s_2 dans T et conserve donc sa position. Dans T', le somme s reste bien avant le sommet s'.
   - Si s est égal à s_1, alors s' apparaît après s_1 dans T. En outre, par hypothèse il n'y a pas d'arête $s_1 \longrightarrow s_2$ donc s' ne peut pas être égal à s_2 : le sommet s' apparaît après s_1 et s_2 et on conclut comme au cas précédent.
   - On conclut symétriquement si s' est égal à s_1 ou à s_2.
8. Si un graphe G possède un chemin hamiltonien, alors ce chemin donne un tri topologique dans lequel chaque sommet est relié au suivant par une arête (question 6).
   Réciproquement, si un tri topologique $s_1, \ldots, s_n$ est tel que pour tout $i \in [1, n[$ on a une arête $s_i \longrightarrow s_{i+1}$, alors il y a un chemin $s_1 \longrightarrow \ldots \longrightarrow s_n$. Par définition d'un tri topologique, chaque sommet s du graphe G apparaît exactement une fois dans la séquence $s_1, \ldots, s_n$, et le chemin précédent est donc hamiltonien.
9. Algo : réaliser un tri topologique, puis vérifier l'existence d'une arête entre chaque paire de sommets consécutifs. C'est effectivement linéaire en la taille du graphe dès lors que le coût de la vérification de la présence d'une arête $s \longrightarrow s'$ est proportionnel dans le pire des cas au degré sortant de s (les deux représentations vues en cours ont bien cette propriété).
10. Pour le tri topologique, on repère un sommet de degré entrant 0, puis on le retire du graphe et on recommence. Ici, il n'y a à chaque étape qu'un unique sommet sélectionnable. Résultat : $H \longrightarrow G \longrightarrow D \longrightarrow D \longrightarrow F \longrightarrow B \longrightarrow A \longrightarrow C \longrightarrow J$.

Correction exercice 3.

1. Spécification : le résultat `r` est la somme d'un segment de `t`, et aucun autre segment n'a une somme supérieure.

$$\left(\exists i, j \in [0, n], \mathtt{r} = \sum_{i \le k < j} \mathtt{t}[k]\right) \wedge \left(\forall i, j \in [0, n], \mathtt{r} \ge \sum_{i \le k < j} \mathtt{t}[k]\right)$$

   Note : si $j < i$ alors $\sum_{i \le k < j} \mathtt{t}[k] = 0$ et la formule reste vraie même sans imposer $i \le j$.

2. Si tous les nombres sont positifs, la somme maximale est la somme de tous les éléments. Si tous les nombres sont négatifs, la somme maximale est 0 (somme de zéro éléments).
3. La boucle interne fait $(n+1) - (\mathtt{lo}+1) = n - \mathtt{lo}$ tours à chaque tour de la boucle externe. Donc nombre total d'appels

$$\sum_{0 \le \mathtt{lo} < n} (n - \mathtt{lo}) = \sum_{1 \le k \le n} k = \frac{n(n+1)}{2} \quad \frac{n^2}{2}$$

4. La fonction ayant elle-même une complexité linéaire, on obtient une complexité totale cubique.

Étape	`lo`	`k`	`sum`	`maxSum`
	0	0	0	0
1	0	0	−2	0
2	0	1	1	1
3	0	2	−1	1
4	0	3	2	2
5	1	1	3	3
6	1	2	1	3
7	1	3	4	4
8	2	2	−2	4
9	2	3	1	4
10	3	3	3	4

5.
6. La boucle interne fait $n - \mathtt{lo}$ accès à chaque tour de la boucle externe, d'où un total de $\frac{n(n+1)}{2} \sim \frac{n^2}{2} = \Theta(n^2)$.
7. La variable `sum` contient la somme des éléments du segment $\mathtt{t}[\mathtt{lo}, \mathtt{k}[$.

$$\mathtt{sum} = \sum_{\mathtt{lo} \le i < \mathtt{k}} \mathtt{t}[i]$$

8. La variable `maxSum` contient la somme maximale parmi les segments $\mathtt{t}[a, b[$ avec $a < \mathtt{lo}$.

$$\left(\exists i, j \in [0, n], i < \mathtt{lo} \wedge \mathtt{maxSum} = \sum_{i \le l < j} \mathtt{t}[l]\right) \wedge \left(\forall i, j \in [0, n], i < \mathtt{lo} \implies \mathtt{maxSum} \ge \sum_{i \le l < j} \mathtt{t}[l]\right)$$

9. La variable `maxSum` contient la somme maximale parmi les segments $\mathtt{t}[a, b[$ avec $a < \mathtt{lo}$ ou $a = \mathtt{lo} \wedge b < \mathtt{k}$.

$$\left(\exists i, j \in [0, n], (i < \mathtt{lo} \vee (i = \mathtt{lo} \wedge j < \mathtt{k})) \wedge \mathtt{maxSum} = \sum_{i \le l < j} \mathtt{t}[l]\right) \wedge \left(\forall i, j \in [0, n], (i < \mathtt{lo} \vee (i = \mathtt{lo} \wedge j < \mathtt{k})) \implies \mathtt{maxSum} \ge \sum_{i \le l < j} \mathtt{t}[l]\right)$$

10.

Étape	`k`	`sum`	`maxSum`
	0	0	0
1	0	2	2
2	1	1	2
3	2	0	2
4	3	3	3
5	4	2	3
6	5	4	4
7	6	0	4
8	7	1	4

11. Exactement n accès (un à chaque indice sans exception).
12. La valeur de la variable `sum` est la plus grande somme d'un suffixe du segment $\mathtt{t}[0, \mathtt{k}[$.

$$(\exists i \in [0, \mathtt{k}], \mathtt{sum} = \sum_{i \le j < \mathtt{k}} \mathtt{t}[j]) \wedge (\forall i \in [0, \mathtt{k}], \mathtt{sum} \ge \sum_{i \le j < \mathtt{k}} \mathtt{t}[j])$$

13. Première remarque : la valeur s' de `sum` après la ligne 6 est une somme d'un suffixe de `t`$[0, k]$ contenant au moins la case `t[k]` (car la valeur s de `sum` au début du tour de boucle est elle-même la somme d'un suffixe de `t`$[0, k[$). Cette somme est par ailleurs maximale : si elle ne l'était pas, alors on contredirait la maximalité de s de `sum`.

    Raisonnement par cas sur la somme maximale S d'un suffixe de `t`$[0, k]$. Il s'agit :

    — soit de la somme du suffixe vide, c'est-à-dire 0,

    — soit de la somme d'un suffixe non vide, c'est-à-dire contenant au moins la dernière case `t[k]`.

    Dans le deuxième cas, cette somme est précisément s'. Dans le premier cas, s' n'est pas strictement supérieure à zéro, et donc la valeur de `sum` à la fin du tour est exactement 0.

    Pour `maxSum`, raisonnons maintenant par cas sur la somme maximale M d'un segment de `t`$[0, k]$.

    — Si cette somme M peut être obtenue dans un segment de `t`$[0, k[$, alors c'est déjà la valeur qu'avait `maxSum` au début du tour. Comme `sum` à la ligne 7 correspond à la somme d'un segment de `t`$[0, k]$, par maximalité on a bien `sum` $\leq M$, et `maxSum` n'est pas modifiée.

    — Sinon, cette somme M est la somme d'un segment incluant l'indice k, c'est-à-dire la somme d'un suffixe de `t`$[0, k]$. Alors la variable `sum`, qui contient la somme maximale d'un tel suffixe, vaut précisément M, et `maxSum` est bien mise à jour de la bonne manière.

14. On réfute ce critère avec un contre-exemple, dans lequel le segment de somme maximale combine des éléments des deux moitiés du tableau : si on prend le tableau | 1 | 1 | et les indices `lo` = 0, `hi` = 2 et `mid` = 1, on a $M_1 = 1$ et $M_2 = 1$, mais la somme maximale cherchée est 2.

15. À nouveau, on donne un contre-exemple, dans laquelle le préfixe maximal déborde de la première moitié. L'exemple donné à la question précédente convient ($P = 1$ alors que le préfixe maximal du segment entier a la somme 2).

16. Dans le segment vide on peut former la somme nulle, et uniquement elle. Toutes les sommes valent donc 0.

17. Les trois premières composantes du quadruplet sont fausses si `t[lo]` est un nombre strictement négatif (car on peut toujours former la somme nulle, qui est plus grande que `t[lo]`). La quatrième composante en revanche est correcte. Pour corriger cela, on peut remplacer les trois premières composantes par `max(0, t[lo])`.

18. On traite chaque composante séparément :

    `q[0]`. Les valeurs `mslo[0]`, `mshi[0]` sont les sommes de segments inclus respectivement dans la moitié gauche et la moitié droite. La valeur `mlso[2] + mshi[1]` est la somme d'un segment obtenu en concaténant un suffixe de la moitié gauche et un préfixe de la moitié droite, qui est bien un segment du tableau complet. Concluons par cas sur la somme maximale M cherchée :

    — si elle est réalisée dans la moitié gauche, alors elle est aussi la somme maximale de `t`$[lo, mid[$, et donc égale à `mslo[0]`,

    — de même, si elle est réalisée dans la moitié droite, alors elle est égale à `mshi[0]`,

    — enfin, si le segment maximal n'est inclus ni d'un côté ni de l'autre, alors il combine un suffixe de la moitié gauche et un préfixe de la moitié droite, et la somme est égale à `mlso[2] + mshi[1]`.

    `q[1]`. Similaire, avec seulement deux cas : le préfixe maximale peut être inclus dans la moitié gauche, ou être constitué de l'intégralité de la moitié gauche plus un préfixe de la moitié droite.

    `q[2]`. Similaire, avec deux cas.

    `q[3]`. Immédiat, car on ne fait qu'ajouter les deux sous-totaux.

19. Lorsque $n < 2$, il n'y a pas d'appel récursif, on compte seulement le coût de base a. Lorsque $n \geq 2$ on compte en plus deux appels récursifs, sur des segments de taille moitié (avec arrondi).

$$\begin{cases} C(0) &= a \\ C(1) &= a \\ C(n) &= a + C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) \end{cases}$$

20. En se restreignant à des puissances de 2 on obtient les équations

$$\begin{cases} C(2^0) &= a \\ C(2^k) &= a + 2C(2^{k-1}) \qquad k > 0 \end{cases}$$

    De la deuxième équation, en divisant des deux côtés par 2^k on déduit

$$\frac{C(2^k)}{2^k} = \frac{a}{2^k} + \frac{C(2^{k-1})}{2^{k-1}} = \sum_{0<l\leq k} \frac{a}{2^l} + C(2^0) = \sum_{0<l\leq k} \frac{a}{2^l} + a = a(1 + \sum_{0<l\leq k} \frac{1}{2^l}) = a(2 - \frac{1}{2^k})$$

    Puis en multipliant par 2^k

$$C(2^k) = a(2 \times 2^k - 1)$$

    Finalement $C(n) \sim 2an = \Theta(n)$. Même ordre de grandeur que Kadane, mais avec une constante nettement plus élevée, du fait du nombre d'opérations couvertes par le a.