Épreuve écrite d'informatique
Second concours 2022 – ENS Lyon

Membre du jury: Loris Marchal

Durée: 3h

Le sujet s'intéressait à la gestion d'un cache, en commençant par l'étude et l'implémentation de stratégies simples (FIFO, LRU) et en étudiant ensuite le rapport de compétitivité entre ces stratégies et un algorithme omniscient optimal. La dernière partie du sujet traitait du tri du tableau avec cache borné pour minimiser les transferts de données entre mémoire et cache.

Cette épreuve a été passée par 15 candidat.

La première partie s'intéressait à des stratégies simples de gestion de cache. On demandait de dérouler les algorithmes proposés sur un exemple simple, de prouver quelques propriétés simples, et de programmer (en pseudo-code ou dans un langage au choix du candidat) une stratégie LRU, d'abord de façon simple en utilisant des tableaux, puis à l'aide d'une liste doublement chaînée, après avoir détaillé les opérations de mise à jour de cette liste. La plupart des candidats ont réussi les questions introductives simples, et une grosse moitié d'entre eux a bien traité l'intégralité de cette partie, qui testait leurs capacités en algorithmique et programmation.

La seconde partie était plus théorique : après un petit exemple pour comprendre ce qu'était une politique omnisciente optimale, on demandait aux candidats de comparer le nombre de chargements de LRU et d'une telle politique optimale. Les candidats étaient guidés pour prouver que LRU ne pouvait pas faire plus de 2 fois plus de chargements que l'optimal. Seuls 5 candidats ont vraiment abordé cette partie, avec plus ou moins de succès ; un seul candidat l'a correctement et entièrement traitée.

La dernière partie étudiait l'adaptation d'un algorithme de tri classique (tri fusion) donné dans l'énoncé, à une machine avec cache de taille bornée. Les deux premières questions, concernant l'analyse de l'algorithme donné, ont été bien traitées par une grosse moitié des candidats, alors que les deux questions suivantes concernant l'adaptation pour un cache borné n'ont presque pas été abordée et jamais avec succès.

Au final, ce sujet a permis de mettre en évidence les candidats possédant à la fois de bonnes bases en algorithmiques et programmation et capables de raisonner sur des objets informatiques.

Pour information, un corrigé du sujet est proposé à la suite de ce rapport. Nous encourageons les candidats à le lire afin de mieux comprendre les attendus de l'épreuve, en particulier en terme de rédaction des algorithmes et des preuves.

Ce sujet comporte 3 parties. La première partie présente le problème et permet de se familiariser avec les notions abordées dans le sujet. Les parties 2 et 3 sont indépendantes entre elles. Il est recommandé de lire l'ensemble du sujet avant de commencer la rédaction. Il est également conseillé de traiter les questions dans l'ordre de l'énoncé. On pourra cependant aborder une question en admettant les résultats des questions précédentes. Les algorithmes demandés seront écrits dans un langage de programmation au choix du candidat ou en pseudo-code.

Préliminaires et définitions

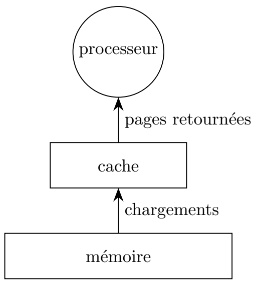
Complexité. Par complexité en temps d'un algorithme $\mathcal{A}$, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de $\mathcal{A}$ dans le pire cas. Le logarithme et l'exponentielle sont considérés comme des opérations élémentaires.

On rappelle la définition de la notation $O(\cdot)$: lorsque la complexité dépend d'un paramètre n, on dit que $\mathcal{A}$ a une complexité en $O(f(n))$ lorsqu'il existe deux constantes k et n_0 telles que la complexité de A est inférieure ou égale à $k \times f(n)$ pour tout $n \geqslant n_0$.

Tableaux. On considère dans ce sujet que les tableaux sont indexés à partir de 0 : un tableau T de m éléments contient les éléments $T[0], \ldots, T[m-1]$.

Définitions. On s'intéresse dans ce sujet au mécanisme de cache présent dans de nombreux systèmes informatiques. On considère ici qu'un **processeur** dispose d'une **mémoire** de grande taille dans laquelle sont stockées des **pages** de données auxquelles on souhaite accéder. Le **chargement** d'une page depuis la mémoire jusqu'au processeur, c'est-à-dire sa lecture dans la mémoire, prend du temps : on cherche à minimiser le nombre de ces chargements. Pour ce faire, on dispose d'un **cache** de pages, qui est une mémoire intermédiaire, de taille limitée (contenant au plus N pages), mais d'accès beaucoup plus rapide. L'algorithme `DemandePage` ci-dessous explicite le processus de chargement des pages. Lorsque le processeur demande une page p, on regarde d'abord si elle est déjà dans le cache : dans ce cas, la page est retournée à l'utilisateur sans aucun chargement. Sinon, on charge la page dans le cache depuis la mémoire (c'est-à-dire qu'on lit la page dans la mémoire et on l'écrit dans le cache), puis la page est retournée à l'utilisateur. On notera qu'une page chargée depuis la mémoire doit toujours être stockée dans le cache.

<u>`DemandePage`(p) :</u>
si p **est dans le cache à la position** i **alors**
  retourner `cache`$[i]$
sinon
  $j \leftarrow$ `ChoisirPosition`()
  `cache`$[j] \leftarrow$ `Charger`(p)
  retourner `cache`$[j]$



On se concentre dans ce sujet au choix de la position j à laquelle est stockée une nouvelle page dans le cache. Tant que le cache n'est pas rempli, on peut sans risque choisir une position qui ne contient pas encore de page. Quand le cache est rempli et qu'on choisit une position j, la page présente à la position j va être **évincée** du cache, c'est-à-dire remplacée par une autre page ; elle ne sera donc plus accessible pour les requêtes futures (il faudra la recharger depuis la mémoire).

Partie 1 Stratégies simples

On considère la stratégie simple suivante pour la fonction `ChoisirPosition`, où $\mod$ désigne l'opérateur modulo. La variable globale i est initialisée avant tout appel à `ChoisirPositionFIFO` et conserve sa valeur courante entre deux appels.

Variable i initialisée à -1

`ChoisirPositionFIFO()` :

$i \leftarrow i + 1 \mod N$

retourner i

Question 1. *On considère un cache de taille $N = 4$. La fonction* `DemandePage` *est appelée successivement avec les pages de la liste suivante $L = [A, B, C, A, D, B, A, E, A, B, F, D]$. On représente l'état du cache après le premier appel sous la forme suivante :*

$$\texttt{cache} = [\underline{A}, \emptyset, \emptyset, \emptyset]$$

où le symbole $\emptyset$ signifie qu'aucune page n'a été chargée à une certaine position du cache. Une page du cache est soulignée si elle a été chargée depuis la mémoire à la dernière étape. Donner l'état du cache après chaque appel à `DemandePage` *pour la liste L, ainsi que le nombre total de chargements de pages dans le cache depuis la mémoire.*

Correction:

après A : $[A, \emptyset, \emptyset, \emptyset]$
après B : $[A, B, \emptyset, \emptyset]$
après C : $[A, B, C, \emptyset]$
après D : $[A, B, C, D]$
après B : $[A, B, C, D]$
après A : $[A, B, C, D]$
après E : $[E, B, C, D]$
après A : $[E, A, C, D]$
après B : $[E, A, B, D]$
après F : $[E, A, B, F]$
après D : $[D, A, B, F]$
en tout : 9 chargements

□

Question 2. *Montrer qu'avec cette stratégie, les pages sont évincées du cache dans leur ordre de chargement depuis la mémoire dans le cache.*

Correction:

On ne considère que les étapes contenant un chargement (et éventuellement une éviction). Les évictions commencent à l'étape N. Pour $k > 0$, la page évincée à l'étape $N + k$ se situe en position $(k - 1) \mod N$. Le dernier chargement à cet emplacement date de l'étape k. Les ordres de chargement et d'éviction sont donc les mêmes. □

On cherche à implémenter une stratégie plus intéressante, qui évince la page X dont la dernière requête `DemandePage`(X) est la plus ancienne (souvent appelée LRU en anglais, pour *Least Recently Used*).

Question 3. *Pour $N = 4$ et la liste L de pages de la question 1, donner l'état du cache après chaque appel à* `DemandePage` *avec cette stratégie, ainsi que le nombre total de chargements de pages depuis la mémoire dans le cache.*

Correction:

après A : $[A, \emptyset, \emptyset, \emptyset]$
après B : $[A, B, \emptyset, \emptyset]$
après C : $[A, B, C, \emptyset]$
après D : $[A, B, C, D]$
après B : $[A, B, C, D]$
après A : $[A, B, C, D]$
après E : $[A, B, E, D]$
après A : $[A, B, E, D]$
après B : $[A, B, E, D]$
après F : $[A, B, E, F]$
après D : $[A, B, D, F]$
au total : 7 chargements □

On définit l'**étape** k comme le $k^{\text{ème}}$ appel à la fonction `DemandePage`. On utilise deux variables globales :

— un entier `étape`, initialisé à 1 et qui désigne l'indice de la prochaine étape
— un tableau `dernierUsage` tel que :

$$\texttt{dernierUsage}[i] = \begin{cases} -1 & \text{si la page } \texttt{cache}[i] \text{ n'a jamais été utilisée} \\ k & \text{si la dernière utilisation de la page } \texttt{cache}[i] \text{ s'est faite à l'étape } k \end{cases}$$

Question 4. *Donner des algorithmes* `DemandePageLRU` *et* `ChoisirPositionLRU` *pour cette stratégie utilisant les variables globales* `étape` *et* `dernierUsage` *(on détaillera leur initialisation). Donner et justifier la complexité d'un appel à* `ChoisirPositionLRU`.

Correction:

Variables globales

`étape` $\leftarrow 0$
pour tout i, `dernierUsage`$[i] \leftarrow -1$

`DemandePageLRU`(p) :

si p **est dans le cache à la position** i **alors**
  `dernierUsage`$[i] \leftarrow$ `étape`
  `étape` $\leftarrow$ `étape` $+ 1$
  retourner `cache`$[i]$
sinon
  $j \leftarrow$ `ChoisirPositionLRU`$()$
  `cache`$[j] \leftarrow$ `Charger`(p)
  `dernierUsage`$[j] \leftarrow$ `étape`
  `étape` $\leftarrow$ `étape` $+ 1$
  retourner `cache`$[j]$

`ChoisirPositionLRU`$()$:

Choisir un indice i atteignant le minimum dans `dernierUsage`$[i]$
retourner i

La complexité de `ChoisirPositionLRU` est en $O(N)$ car il faut chercher le minimum dans un tableau de taille N. La complexité de `DemandePageLRU` est donc également linéaire en la taille du cache : $O(N)$. □

On se concentre dans la fin de cette partie sur le cas où le cache est plein (pour tout i, `dernierUsage`$[i] \geqslant 0$). On considère le graphe dirigé constitué d'une chaîne dont les sommets sont les positions i du cache, et où il y a une arête $i \to j$ si

— `dernierUsage[i]` $<$ `dernierUsage[j]`,

— et pour tout position k différente de i et j, `dernierUsage[k]` $<$ `dernierUsage[i]` ou `dernierUsage[k]` $>$ `dernierUsage[j]`.

Question 5. *On considère le cache $[A, B, C, D]$ et la chaîne suivante :*

$$\begin{array}{c} 0 \\ (\textit{page } A) \end{array} \longrightarrow \begin{array}{c} 2 \\ (\textit{page } C) \end{array} \longrightarrow \begin{array}{c} 1 \\ (\textit{page } B) \end{array} \longrightarrow \begin{array}{c} 3 \\ (\textit{page } D) \end{array}$$

Comment cette chaîne et ce cache seraient-ils modifiés lors d'une requête à la page B ? lors d'une requête à la page E ?

Correction:

Lors de la requête à B, celui-ci passe en fin de chaîne (pas de modification du cache) :

$$(A) \to 2(C) \to 3(D) \to 1(B)$$

Lors de la requête à E, page en tête de la chaîne est évincée et E est rajouté en fin de chaîne (et remplace A en position 0) :

$$2(C) \to 1(B) \to 3(D) \to 0(E)$$

□

On choisit de représenter la chaîne précédente par deux tableaux en variables globales :

— `suivant`$[i]$ donne l'indice suivant i dans la chaîne (ou vaut -1 si i est le dernier élément de la chaîne)

— `précédent`$[i]$ donne l'indice précédent i dans la chaîne (ou vaut -1 si i est le premier élément de la chaîne).

On ajoute également deux variables globales : `débutChaine` et `finChaîne` contenant les indices du premier et dernier sommet de la chaîne.

La chaîne initiale de la question 5 est donc représentée par :

$$\begin{array}{rcl} \texttt{suivant} & = & [2, 3, 1, -1] \\ \texttt{précédent} & = & [-1, 2, 0, 1] \\ \texttt{débutChaîne} = 0 & & \texttt{finChaîne} = 3 \end{array}$$

Question 6. *Donner des algorithmes* `DemandePageLRU2` *et* `ChoisirPositionLRU2` *utilisant la représentation précédente de la chaîne. Justifier leur complexité.*

Correction:

`MettreEnTete`(j) :

$s \leftarrow$ `suivant`$[j]$, $q \leftarrow$ `précédent`$[j]$
`suivant`$[q] \leftarrow s$, `precedent`$[s] \leftarrow q$
`précédent`[`débutChaîne`] $\leftarrow j$
`suivant`$[j] \leftarrow$ `débutChaîne`
`précédent`$[j] \leftarrow -1$
`débutChaîne` $\leftarrow j$

`DemandePageLRU2`(p) :

si p **est dans le cache à la position** i **alors**
    `MettreEnTête`(i)
    retourner `cache`$[i]$
sinon
    $j \leftarrow$ `ChoisirPositionLRU2`$()$
    `cache`$[j] \leftarrow$ `Charger`(p)
    `MettreEnTête`(j)
    retourner `cache`$[j]$

`ChoisirPositionLRU2`$()$:

retourner `finChaîne`

La fonction `MettreEnTête` effectue un nombre constant d'opérations à chaque appel, de même que la fonction `ChoisirPositionLRU2`. Leur complexité est donc en $O(1)$, de même que la complexité de `DemandePageLRU2`. □

Partie 2 Comparaison à un algorithme omniscient

On s'intéresse dans cette partie à un algorithme optimal nommé OPT qui connaît la séquence des requêtes futures à des pages de la mémoire et décide en conséquence des pages à évincer du cache afin de minimiser le nombre de chargements.

Question 7. *Pour $N = 4$ et la liste L de page de la question 1, donner l'état du cache à chaque étape ainsi que le nombre total de chargements depuis la mémoire dans le cache d'une telle stratégie optimale.*

Correction:

après A : $[A, \emptyset, \emptyset, \emptyset]$
après B : $[A, B, \emptyset, \emptyset]$
après C : $[A, B, C, \emptyset]$
après D : $[A, B, C, D]$
après B : $[A, B, C, D]$
après A : $[A, B, C, D]$
après E : $[A, B, E, D]$
après A : $[A, B, E, D]$
après B : $[A, B, E, D]$
après F : $[A, B, F, D]$
après D : $[A, B, F, D]$

au total : 6 chargements □

Dans la suite de cette partie, on considère une séquence S de requêtes ainsi qu'une stratégie OPT qui est optimale sur S. On cherche à comparer le nombre de chargements de la stratégie LRU (présentée à la question 3) et celui de OPT sur cette même séquence S. On considère ici que les deux stratégies comparées n'ont pas nécessairement la même taille de cache : N est la taille du cache pour LRU et N' celle pour la stratégie OPT. On suppose $N \geqslant N'$. Enfin, on considère qu'avant d'aborder la séquence S, les caches des deux stratégies sont pleins : ils contiennent déjà des pages (mais on ne fait aucune hypothèse sur les pages qu'ils contiennent).

On appelle sous-séquence de S une séquence S^* de requêtes qui apparaissent consécutivement dans S. Pour toute sous-séquence S^* de S, on note $C(S^*)$ (respectivement $C'(S^*)$) le nombre de chargements depuis la mémoire dans le cache fait par LRU (resp. OPT) pendant le traitement de S^*.

On partititionne S en $k+1$ sous-séquences S_0, S_1, ...,S_k telles que : $C(S_0) \leqslant N$, et pour $i \geqslant 1$, $C(S_i) = N$. Autrement dit, chacune des sous-séquences constituant S cause exactement N chargements pour la stratégie LRU, sauf la première qui en cause au plus N.

Question 8. *On considère une sous-séquence S_i, $i > 0$ de S ainsi que la page r de la requête immédiatement avant S_i dans S.*

(a) On suppose que pendant qu'il traite S_i, LRU ne charge pas dans le cache deux fois la même page et ne recharge pas r. Montrer que

$$C'(S_i) \geqslant C(S_i) - N' + 1$$

(b) Montrer que si LRU charge deux fois la même page dans le cache ou recharge r dans le cache pendant S_i, l'inégalité ci-dessus est encore vérifiée.

Correction:

Dans le premier cas, on sait que LRU charge $C(S_i)$ pages distinctes et différentes de r. Pour arriver à fournir ces pages, OPT peut compter sur son cache, mais dont une case au moins est occupée par r. Il fera donc au moins $C(S_i) - (N' - 1)$ chargements.

Si LRU charge deux fois la même page, ça veut dire qu'entre les 2 chargements, il a vidé son cache, donc qu'il a y a au moins N requêtes différentes dans S_i entre ces deux chargements. S'il recharge r, alors de même il doit y avoir N requêtes différentes entre le début de S_i et le rechargement de r. Dans les deux cas, il y a au moins $N + 1$ requêtes différentes dans S_i. OPT devra donc faire au moins $N + 1 - N'$ chargements. Comme $C(S_i) \leqslant N$, on peut conclure. □

Question 9. *Montrer que $C'(S_0) \geqslant C(S_0) - N'$.*

Correction:

Dans S_0, il y a au moins $C'(S_0)$ requêtes. Soit toutes ces requêtes sont distinctes, auquel cas, le cache de OPT lui permet d'économiser au plus N' chargements, et on a l'inégalité. Soit il y a 2 requêtes pour la même page, et alors comme à la dernière question, LRU a touché à toutes les pages de son cache pendant S_0, et donc il y a au moins $N \geqslant C(S_0)$ requêtes dans S_0, on conclut comme dans le premier cas. □

Question 10. *Montrer que*

$$C(S) \leqslant \frac{N}{N - N' + 1} C'(S) + Z$$

où Z est un terme ne dépendant pas de la séquence S ou de ses sous-séquences.

Correction:

Pour chaque séquence $S_i, i > 0$, on applique le résultat de la question 8. On a donc :

$$C'(S_i) \geqslant C(S_i) - N' + 1 = N - N' + 1 = \frac{N - N' + 1}{N} C(S_i)$$

soit

$$\frac{C(S_i)}{C'(S_i)} \leqslant \frac{N}{N - N' + 1}$$

D'après la dernière question, on a $C'(S_0) \geqslant C(S_0) - N'$. On distingue deux cas :

— $C(S_0) - N' < 0$, alors

$$C'(S_0) \geqslant 0 > \frac{N - N' + 1}{N}(C(S_0) - N')$$

— $C(S_0) - N' < 0$, alors comme $\frac{N-N'+1}{N} \leqslant 1$, on a aussi

$$C'(S_0) \geqslant \frac{N - N' + 1}{N}(C(S_0) - N')$$

Ainsi :

$$\frac{C(S_0) - N'}{C'(S_0)} \leqslant \frac{N - N' + 1}{N}$$

En appliquant l'inégalité donnée dans la question à $C(S) - N' = (C(S_0) - N') + \sum_{i=1}^{k} C(S_i)$ et $C'(S) = \sum_{i=0}^{k} C'(S_i)$, on obtient :

$$\frac{C(S) - N'}{C'(S)} \leqslant \frac{N - N' + 1}{N}$$

et donc

$$C(S) \leqslant \frac{N - N' + 1}{N} C'(S) + N'$$

. □

On cherche maintenant à montrer qu'il existe des séquences de pages où la borne de la question précédente est atteinte (à un facteur additif près). On note K (respectivement K') l'ensemble des pages présentes initialement dans le cache de LRU (resp. OPT). On considère la séquence de pages S décrite par ces deux étapes :

(a) La séquence commence par $N - N' + 1$ pages distinctes qui ne sont ni dans K ni dans K'. On note S_a l'ensemble de ces pages.

(b) La séquence comporte ensuite $N' - 1$ pages définies de la façon suivante : à chaque instant, on choisit une page de $K' \cup S_a$ qui n'est pas dans le cache de LRU.

Question 11. *Justifier que pour l'étape (b), on peut toujours trouver une page de $K' \cup S_a$ qui n'est pas dans le cache de LRU.*

Correction:

On travaille sur $N + 1$ pages alors que le cache de A est de taille N : il y a toujours au moins une page évincée, c'est celle qu'on choisit comme prochaine page accédée. □

Question 12. *Calculer le ratio $C(S)/C'(S)$ pour la séquence S ainsi construite.*

Correction:

LRU fait un chargement pour chaque requête, donc au total $(N - N' + 1) + (N' - 1) = N$. Au contraire, l'optimal garde à la fin de la première étape uniquement les $N' - 1$ pages de la deuxième étape (plus la dernière page de la première étape). Donc il ne fait un chargement que pour les pages de la première étape, au total $N - N' + 1$ chargements.

Pour la séquence construite, $C(S)/C'(S) = \frac{N}{N-N'+1}$. □

Partie 3 Tri d'un tableau avec cache borné

On considère dans cette partie des grands tableaux d'entiers que l'on cherche à trier. On désigne par $T[i]$ l'élément d'indice i du tableau T, le premier élément étant indicé par $i = 0$. On appelle $T[i, j]$ le tableau constitué des éléments de T d'indices i à j inclus, c'est-à-dire le tableau $[T[i], T[i+1], \ldots, T[j]]$. On considère l'algorithme `TriFusion` de tri d'un tableau T de $M = 2^m$ éléments décrit ci-dessous :

```
Fusion(A, B) :
M_A ← taille(A)
M_B ← taille(B)
Soit C un tableau de taille M_A + M_B
i ← 0, j ← 0
tant que i < M_A et j < M_B faire
    si A[i] < B[j] alors
        C[i + j] ← A[i], i ← i + 1
    sinon
        C[i + j] ← B[j], j ← j + 1
tant que i < M_A faire
    C[i + j] ← A[i], i ← i + 1
tant que j < M_B faire
    C[i + j] ← B[j], j ← j + 1
retourner C
```

```
TriFusion(T, M, p) :
si M = 1 alors
    retourner T
sinon
    A ← TriFusion(T[0, M/2 − 1] , M/2, p + 1)
    B ← TriFusion(T[M/2, M − 1], M/2, p + 1)
    retourner Fusion(A, B)
```

L'appel initial est `TriFusion`$(T, M, 1)$. Le paramètre p désigne la **profondeur** de l'appel : il vaut 1 pour l'appel initial, puis est incrémenté à chaque nouvel appel récursif.

Question 13. *Montrer que si A et B sont deux tableaux de taille M triés par ordre croissant,* `Fusion`(A, B) *renvoie un tableau trié par ordre croissant des éléments de A et B, et donner sa complexité en nombre de comparaisons.*

<u>**Correction:**</u>
La preuve de correction, non détaillée complètement ici, peut s'appuyer sur l'invariant suivant : à tout instant de l'exécution de l'algorithme, le sous-tableau $C[i+j]$ contient les $i+j$ plus petits élements de A et B triés par ordre croissant.

Tant que A et B ne sont pas vides l'un ou l'autre, on remplit le tableau résultat avec l'élément le plus petit non encore recopié de A et B. Puis, si A (resp. B) est vide, on termine en recopiant les éléments de B (resp. de A) dans l'ordre. À chaque comparaison, un élément est recopié dans le résultat, donc il y a au plus autant de comparaisons que d'éléments de $A \cup B$. □

Question 14. *Lors de l'exécution de* `TriFusion`$(T, M, 1)$ *avec $M = 2^m$, on veut connaître le nombre total d'appels récursifs à* `TriFusion` *pour chaque profondeur p atteinte (c'est-à-dire les appels* `TriFusion`$(*, *, p)$*), ainsi que la taille du tableau d'entrée pour ces appels. Donner les valeurs correspondant aux points d'interrogation dans le tableau suivant :*

Profondeur	*Nombre d'appels*	*Taille du tableau*
1	*1*	M
2	*2*	$M/2$
...	...	...
ℓ	*?*	*?*
...	...	...
?	*?*	*1*

En déduire le nombre total de comparaisons effectuées par `TriFusion`(T, M).

Correction:

Profondeur	Nombre d'appels	Taille du tableau
1	1	M
2	2	$M/2$
...	...	...
ℓ	2^ℓ	$M/2^\ell$
...	...	...
$\log_2 M = m$	$2^m = M$	1

□

On souhaite adapter l'algorithme `TriFusion` pour qu'il fonctionne efficacement sur un système avec cache, comme présenté dans la partie 1. On suppose qu'une page peut contenir Q éléments du tableau à trier (et le cache peut contenir N pages, comme précédemment). Initialement, le tableau à trier est en mémoire (et pas dans le cache). À la fin du traitement, le tableau trié doit également se trouver en mémoire. Une différence avec la partie 1 est que lors de l'éviction d'une page qui a été modifiée depuis son chargement dans le cache, celle-ci devra être **réécrite en mémoire**. On s'attachera donc à minimiser le nombre total de chargements de pages dans le cache et d'écriture de pages dans la mémoire, qu'on appelle le **volume d'entrées/sorties**.

On considère l'adaptation suivante de l'algorithme `TriFusion` :

(a) Le cas de base considère un tableau de taille NQ (et non plus 1). On commence par charger tout le tableau dans le cache. On le trie (par exemple en utilisant la fonction `TriFusion`), puis on écrit le résultat en mémoire. On admettra que cette étape nécessite de charger exactement N pages dans le cache et d'écrire exactement N pages en mémoire.

(b) Lorsque le tableau est d'une taille $M > NQ$, on remplace la première boucle ≪ tant que ≫ de la procédure `Fusion` de la façon suivante :

$i_{\max} \leftarrow 0, j_{\max} \leftarrow 0, c_{\min} \leftarrow 0$
Réserver une page du cache pour écrire les éléments de $C[c_{\min}, c_{\min} + Q - 1]$
tant que $i < M_A$ **et** $j < M_B$ **faire**
  si i==$i_{\max}$ **alors**
    charger $A[i, i+Q]$ dans une page de cache
    $i_{\max} = i_{\max}$ +Q
  si j==$j_{\max}$ **alors**
    charger $B[j, j+Q]$ dans une page de cache
    $j_{\max} = j_{\max}$ +Q
  si $A[i] < B[j]$ **alors**
    $C[i+j] \leftarrow A[i]$, $i \leftarrow i+1$
  sinon
    $C[i+j] \leftarrow B[j]$, $j \leftarrow j+1$
  si i+j - $c_{\min}$ **== Q alors**
    écrire la page $C[c_{\min}, i+j]$ en mémoire
    $c_{\min} \leftarrow i+j$
    Réserver une page du cache pour écrire les éléments de $C[c_{\min}, c_{\min}+Q-1]$

On continue comme dans la version initiale en recopiant les éléments manquants de A ou B, en prenant garde de ne charger qu'une seule fois dans le cache chaque page d'éléments (et de n'écrire qu'une seule fois dans la mémoire chaque page d'éléments).

Question 15. *On suppose que $N = 2^n$ et $Q = 2^q$. Donner le nombres d'éléments chargés dans le cache et écrits en mémoire pour l'ensemble des appels récursifs à la profondeur p qui* *apparaissent lors de l'exécution de l'algorithme* `TriFusion` *adapté. En déduire le nombre total d'entrées/sorties de l'algorithme* `TriFusion` *adapté.*

Correction:

Pour un appel à la fusion de 2 tableaux, chaque page de A et B est chargée exactement une fois, et chaque page de C est écrite une seule fois. Le nombre d'entrée sortie est donc $2 \times |C|$.

À une profondeur p, pour chacun des 2^p appels, le tableau est de taille $M/2^p$. Le nombre total d'entrée sortie pour la profondeur p est donc $2^p \times 2 \times M/2^p = 2M$.

La profondeur maximale est atteinte pour $M/2^p_{\max} = NQ$ donc pour $p_{\max} = \log_2 M/NQ = \log_2 2^m/2^(n+q) = m - n - q$. Le nombre d'entrée/sortie pour la phase récursive est donc $(m - n - q - 1) \times M$.

Le cas de base permet de lire/écrire une seule fois le tableau, ce qui fait $2NQ$ lectures (pour un appel). Il y a $2^{p_{\max}} = M/NQ$ appels au cas de base, ce qui fait donc $2M$ lectures/écritures.

Au total, on a donc $(m - n - q + 1) \times M$ lectures/écritures. □

Question 16. *Quel est le nombre de pages du cache utilisées à tout instant par l'algorithme précédent dans le cas récursif (cas (b) ci-dessus) ? Proposer une optimisation de l'algorithme utilisant mieux le cache, et calculer le nombre total d'entrées/sorties obtenu.*

Correction:

On n'utilise à tout instant que 3 pages du cache au maximum lors de la fusion : une pour des éléments de A, une pour des éléments de B, une pour des éléments de C.

Optimisation possible : On pourrait utiliser les N pages en faisant une fusion de N sous-tableau. On obtient un arbre de fusion d'arité N, et donc de hauteur réduite, ce qui réduit le nombre total d'entrées/sorties. Par contre, il faut avoir une structure de données efficaces pour trouver (et maintenir) le minimum de N valeurs. On peut par exemple utiliser un tas binaire.□

⋆ ⋆ ⋆