ÉCOLE NORMALE SUPÉRIEURE DE LYON

Concours d'admission session 2022

Filière universitaire : Second concours

COMPOSITION D'INFORMATIQUE

Durée : 3 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.

⋆ ⋆ ⋆

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

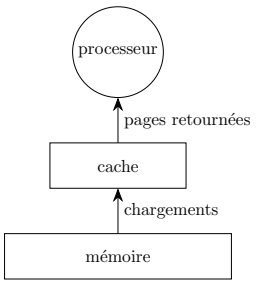
Complexité. Par complexité en temps d'un algorithme $\mathcal{A}$, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de $\mathcal{A}$ dans le pire cas. Le logarithme et l'exponentielle sont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

```
DemandePage(p) :
si p est dans le cache à la position i alors
    retourner cache[i]
sinon
    j ← ChoisirPosition()
    cache[j] ← 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.*

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

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.*

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`*.*

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 :*

$$\underset{(page\ A)}{0} \longrightarrow \underset{(page\ C)}{2} \longrightarrow \underset{(page\ B)}{1} \longrightarrow \underset{(page\ D)}{3}$$

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 ?

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 :

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

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

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.*

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.

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

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.

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.*

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

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.*

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).

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.*

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.*

⋆ ⋆ ⋆