Tris adaptatifs et algorithme TimSort

Le jury attend une présentation de 35 minutes s'appuyant sur ce sujet, en forme de cours, pédagogique, structurée et évitant la paraphrase. Le texte se conclut par des pistes de réflexion facultatives dont vous pouvez vous saisir ; au-delà de ces pistes proposées, toute initiative personnelle pertinente est appréciée. Vous n'êtes pas obligé de traiter le texte dans son intégralité, mais si seule une partie est traitée elle doit l'être de manière particulièrement approfondie.

L'exposé doit intégrer une (ou plusieurs) illustrations informatiques. Il doit également contenir une discussion autour d'une dimension éthique, sociétale, environnementale, économique ou juridique en lien avec le texte ; une des pistes de réflexion est spécifiquement conçue à cet effet.

1 Contexte

Il y a quelques années, la plupart des langages de programmation utilisaient des variantes de l'algorithme de tri rapide (QuickSort) comme implantation de leur tri par comparaisons standard : QuickSort a une très bonne complexité moyenne, est en place, et, en choisissant le pivot aléatoirement, on se ramène au cas moyen quelque soit le tableau de départ.

Ces dernières années, certains langages populaires comme Python et Java on changé d'algorithme pour trier leurs tableaux d'objets, afin de s'adapter à la structure typique observée des entrées. Tim Peters, ingénieur ayant fortement contribué au développement de CPython, l'implantation de référence du langage Python en langage `C` a proposé une nouvelle solution pour trier des données par comparaisons au début des années 2000. A propos de son nouvel algorithme, appelé TimSort, il écrit[1] : "Je crois que les listes sont très souvent partiellement ordonnées dans la vrai vie, et c'est le meilleur argument en faveur de TimSort".

Son algorithme est basé sur une décomposition du tableau en séquences monotones, puis en des fusions astucieuses de séquences ordonnées. Dans ce document nous allons voir plusieurs solutions basées sur ce principe, ainsi que deux façons de

[1] "I believe that lists very often do have exploitable partial order in real life, and this is the strongest argument in favor of timsort"

quantifier le désordre dans un tableau afin de pouvoir évaluer leur efficacité pour des tableaux partiellement ordonnés.

2 Décomposition en séquences croissantes maximales

La *concaténation* de deux tableaux $X = (x_0, \ldots, x_{n-1})$ et $Y = (y_0, \ldots, y_{m-1})$ est le tableau $X \cdot Y$ de longueur $n+m$ donné par $X \cdot Y = (x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})$.

Dans toute la suite nous considérerons que les données à trier sont des tableaux ou des tableaux redimensionnables, et on ne considère que des tris par comparaisons. Tout tableau T de longueur $|T| \geq 1$ peut être décomposé en $\rho \geq 1$ séquences croissantes maximales (SCM), qui sont des tableaux $S_1, \ldots, S_\rho$ dont la concaténation fait T: $T = S_1 \cdot S_2 \cdots S_\rho$, qui sont tous croissants, et tels que pour tout $i \in \{1, \ldots, \rho - 1\}$, le dernier élément de S_i est strictement plus grand que le premier élément de S_{i+1}. Un exemple est donné dans la figure 1.

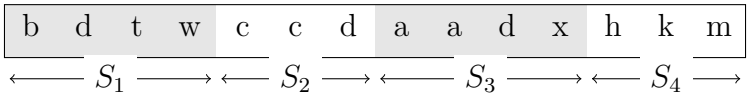

Figure 1: Exemple de la décomposition d'un tableau en 4 SCM.

Cette décomposition est unique, et on va utiliser le nombre de SCM, noté ρ dans toute la suite, comme une première mesure d'à quel point T est ordonné ou non : plus ρ est petit, plus on considérera T comme déjà partiellement ordonné. On verra une mesure plus fine dans un second temps.

On peut facilement calculer la décomposition en SCM d'un tableau en temps linéaire. Chacune des séquences croissantes étant déjà triée, on peut combiner deux telles séquences de la même façon que pour l'algorithme de Tri Fusion (MergeSort) : deux tableaux déjà ordonnés S et S' peuvent être fusionnés pour donner un tableau trié contenant tous les éléments de S et de S' (avec multiplicités) en utilisant moins de $|S| + |S'|$ comparaisons.[2] On notera $S \oplus S'$ le résultat de la fusion de S et de S'. Sur la figure 1 on a par exemple $S_2 \oplus S_3 = [a, a, c, c, d, d, x]$.

3 Premier cadre d'étude

3.1 Arbre de fusions

Notre premier cadre d'étude est donc le suivant : on cherche un algorithme efficace pour trier un tableau à partir de sa décomposition en SCM. Un tel algorithme doit :

[2]La vraie borne sur le nombre de comparaisons est $|S| + |S'| - 1$, mais on utilisera $|S| + |S'|$ dans toute la suite pour simplifier la présentation.

1. commencer par décomposer le tableaux en SCM ;
2. fusionner successivement des séquences croissantes, le coût en nombre de comparaisons de la fusion $S \oplus S'$ de deux séquences S et S' étant évalué à $|S| + |S'|$;
3. s'arrêter quand il n'y a plus qu'une séquence : c'est le tableau trié.

On peut représenter les fusions effectuées par un tel algorithme par un *arbre des fusions*, qui est un arbre binaire complet (chaque nœud interne a deux fils) dont les feuilles sont les SCM, et tel que deux séquences fusionnées sont les deux enfants d'un même père.

Pour un arbre des fusions $\mathcal{A}$, dont les SCM sont $S_1, \ldots, S_\rho$, le son *coût total* est évalué à

$$c(\mathcal{A}) = \sum_{i=1}^{\rho} h(S_i)\,|S_i|, \tag{1}$$

où $h(S_i)$ est la hauteur de la feuille S_i, c'est-à-dire sa distance à la racine. Cela correspond bien à la somme des coûts de toutes les fusions de l'arbre. Deux exemples d'arbres des fusions avec des coûts différents sont donnés figure 2.

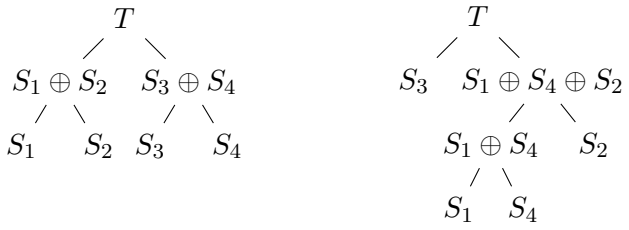

Figure 2: Deux exemples d'arbres des fusions pour les quatre SMC de l'exemple de la figure 1. Les SMC S_1, S_2, S_3 et S_4 étant de longueurs respectives 4, 3, 4 et 3, le coût total de l'arbre de gauche est évalué à 28, alors que celui de l'arbre de droite est évalué à 31.

3.2 Algorithmes pour le premier cadre d'étude

Le premier algorithme pour le problème ainsi posé a été proposé par Knuth, sous le nom de *Tri Fusion Naturel*. Il s'agit de prendre un arbre des fusions le plus équilibré possible, comme l'arbre de gauche de la figure 2. Toutes les feuilles sont à même hauteur ou presque (écart d'au plus un) qui est à peu près le logarithme du nombre de SCM, que l'on a noté ρ. On peut montrer que la complexité de l'algorithme est en $\mathcal{O}(n + n \log \rho)$. C'est un premier exemple de ce que l'on appelle un *tri adaptatif* pour le paramètre ρ : on peut montrer que cette complexité est

optimale pour des tris par comparaisons, à la constante multiplicative cachée dans la notation $\mathcal{O}$ près.

On peut également observer que l'équation (1) est la même que le coût de l'encodage par un code préfixe d'un texte avec ρ caractères différents, dont les nombres d'occurrences sont $|S_1|, |S_2|, \ldots, |S_\rho|$. On sait qu'un codage qui minimise $c(\mathcal{A})$ est donné par un arbre de Huffman. Appliquer la construction de Huffman aux SCM, en prenant leurs longueurs comme poids, permet donc de trouver un arbre des fusions optimal. Cet arbre peut se calculer en $\mathcal{O}(\rho \log \rho)$ et on sait, comme on a un résultat au moins aussi bon que le Tri Fusion Naturel, que la complexité est aussi en $\mathcal{O}(n + n \log \rho)$: elle ne peut être meilleure que le Tri Fusion Naturel que d'une constante multiplicative, par optimalité de l'algorithme, avec ce choix de mesure de désordre.

3.3 Entropie binaire des SCM

Considérons un tableau T de longueur $n \geq 8$ dont la décomposition donne une SCM de longueur $n-7$ et 7 SCM de longueur 1. On a donc $\rho = 8$ et l'algorithme de Tri Fusion Naturel va avoir un arbre des fusions parfaitement équilibré de hauteur 3. Le coût de cet arbre est de $3n$ comparaisons. Si on utilise en revanche un arbre de Huffman sur les longueurs des SCM, on trouve un coût de $n+20$ comparaisons : asymptotiquement il y a presque 3 fois moins de comparaisons.

On peut généraliser la construction ci-dessus pour obtenir une suite de tableaux de taille n tel que le tri fusion naturel utilise $\Theta(n \log n)$ comparaisons, alors qu'utiliser un arbre de Huffman pour les fusions utilise $\Theta(n)$ comparaisons. Pour distinguer les deux algorithmes, il faut donc affiner notre mesure de désordre : juste compter le nombre de SCM n'est pas assez précis.

Pour aller plus loin, on s'inspire de la théorie de l'information et des mesures utilisées pour la compression sans perte. Pour des SCM $\mathcal{S} = (S_1, \ldots, S_\rho)$ dont les longueurs ont pour somme n, on définit l'*entropie binaire* $\mathcal{H}(\mathcal{S})$ de $\mathcal{S}$ par :

$$\mathcal{H}(\mathcal{S}) = -\sum_{i=1}^{\rho} \frac{|S_i|}{n} \log_2 \left(\frac{|S_i|}{n} \right) . \tag{2}$$

On peut montrer que cette fonction est maximale quand toutes les SCM ont même longueur $\frac{n}{\rho}$ (ou à peu près si $\frac{n}{\rho}$ n'est pas un entier). Pour une telle suite de SCM $\mathcal{S}$ on a alors :

$$\mathcal{H}(\mathcal{S}) = -\sum_{i=1}^{\rho} \frac{1}{\rho} \log_2 \frac{1}{\rho} = \log_2 \rho. \tag{3}$$

On peut maintenant utiliser les résultats de théorie de l'information sur l'entropie et sur l'algorithme de Huffman. On a déjà vu que l'arbre des fusions donné par le

procédé de Huffman est optimal pour l'étape de fusion des SCM. Si $\mathcal{A}_H$ est un tel arbre de Huffman pour $\mathcal{S}$, les résultats de compression nous donnent les inégalités suivantes :

$$n\mathcal{H}(\mathcal{S}) \leq c(\mathcal{A}_H) \leq n\mathcal{H}(\mathcal{S}) + n. \tag{4}$$

On obtient donc une borne inférieure en $\Omega(n\mathcal{H}(\mathcal{S}))$ pour trier par comparaisons dans notre cadre d'étude, si on mesure le désordre par $\mathcal{H}(\mathcal{S})$ et non plus le nombre de SCM. Elle est plus précise qu'avec juste le paramètre ρ d'après l'équation (3) ; elle tient compte de la distribution des longueurs des SCM et non plus seulement de leur nombre.

Remarque : pour construire l'arbre de Huffman, il faut une complexité de $\mathcal{O}(\rho \log \rho)$, en utilisant par exemple un tas. Cela peut devenir l'étape limitante s'il y beaucoup de SCM et une entropie faible. Mais la principale raison pour laquelle cet algorithme n'est pas utilisé en pratique n'est pas celle-là, elle est détaillée dans les partie suivantes.

4 Second cadre d'étude

Un des problèmes de l'algorithme qui suit le procédé de Huffman est qu'il fusionne des SCM qui peuvent être éloignées dans le tableau initial. Cela est problématique pour des très grands tableaux, à cause de la gestion de la mémoire. On ajoute donc une contrainte à notre cadre d'étude : "on ne peut fusionner que deux séquences consécutives". Sur un arbre des fusions, cela veut dire qu'on peut imposer aux feuilles d'apparaître dans l'ordre de gauche à droite (ou plus précisément, dans un parcours en profondeur préfixe où on prend le sous-arbre gauche avant le sous-arbre droit). Dans la figure 2, l'arbre de gauche vérifie cette contrainte, mais pas l'arbre de droite où on doit commencer par fusionner la SCM la plus à gauche avec celle la plus à droite.

4.1 Arbre des fusions optimal pour le second cas d'étude

On considère un arbre des fusions $\mathcal{A}$ des SCM $S_1, \ldots, S_\rho$ qui vérifie notre nouvelle contrainte. La dernière fusion à effectuer est à la racine et coûte nécessairement n comparaisons (tous les éléments sont soit dans le sous-arbre gauche, soit dans le sous-arbre droit), plus le coût optimal à gauche et le coût optimal à droite. On peut donc récursivement calculer le coût optimal $C_{\text{opt}}(\mathcal{S})$ d'une suite de SCM en

utilisant la formule récursive :

$$C_{\text{opt}}(S_i, \ldots, S_j) = \begin{cases} 0 \text{ si } i = j \\ \sum_{k=i}^{j} |S_k| + \min_{k \in \{i,\ldots,j\}} \left[C_{\text{opt}}(S_i, \ldots, S_k) + C_{\text{opt}}(S_k, \ldots, S_j) \right] \text{ sinon.} \end{cases} \tag{5}$$

Malheureusement, même en utilisant des techniques algorithmiques classiques pour calculer ce type de formules, cela semble hors d'atteinte d'avoir un algorithme linéaire en n, ce qui est nécessaire pour être performant.

4.2 Un algorithme glouton

On cherche donc une solution plus rapide, quitte à perdre un peu en nombre de comparaisons. On va utiliser un algorithme glouton qui à chaque étape fusionne les deux séquences consécutives qui minimisent la somme de leurs longueurs (la fusion qui coûte le moins cher), jusqu'à ce que tout soit fusionné. Cet algorithme ne produit pas toujours l'arbre des fusions optimal, comme illustré figure 3.

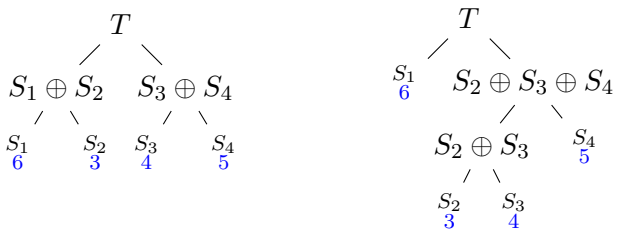

Figure 3: A gauche, l'arbre optimal pour le second cadre, pour des SCM de longueurs 6, 3, 4 et 5, dans cet ordre. À droite, l'arbre produit par l'algorithme glouton. Le premier a un coût de 36, le second a un coût de 37.

On peut cependant démontrer (ce n'est pas du tout immédiat) que l'algorithme glouton est une 2-approximation : son coût est toujours inférieur à deux fois le coût de l'optimal. Cela en fait donc un procédé utilisable, si on structure correctement ses données pour faire le calcul de l'arbre en temps linéaire en n.

On peut également montrer que l'algorithme glouton a une complexité en $\mathcal{O}(n \log \rho + n)$, si on a bien calculé l'arbre en $\mathcal{O}(n)$. En effet, si $\rho \geq 2$ (le cas $\rho = 1$ est immédiat), quand on somme tous les $|S_i| + |S_{i+1}|$ pour $i \in \{1, \ldots, \rho - 1\}$ on obtient moins de $2n$, car chaque SCM intervient dans au plus deux telles sommes. Comme il y a $\rho - 1$ telles paires consécutives, cela signifie que la paire qui minimise cette somme coûte au plus $\frac{2n}{\rho-1}$. Une fois fusionnée par la stratégie gloutonne, on se retrouve avec $\rho - 1$ séquences de somme totale n, et avec le même raisonnement, le coût de la fusion suivante est au plus $\frac{2n}{\rho-2}$, etc. Ainsi le coût total de l'arbre des

fusions de la stratégie gloutonne est au plus :

$$\frac{2n}{\rho-1}+\frac{2n}{\rho-2}+\ldots+\frac{2n}{1}=2n\sum_{i=1}^{\rho-1}\frac{1}{i}=\mathcal{O}(n\log\rho), \tag{6}$$

en utilisant le résultat sur la série harmonique : $\sum_{i=1}^{k}\frac{1}{i}\leq\ln k+1$.

5 Algorithme TimSort

Au début des années 2000, Tim Peters alors ingénieur développeur pour Python, a changé l'algorithme de tri de CPython, pour y intégrer les idées développées dans les parties précédentes. Il a créé un nouvel algorithme, baptisé plus tard TimSort, qui utilise également des fusions de séquences croissantes consécutives. Il ne s'agit cependant pas de la stratégie gloutonne présentée juste avant, l'idée est de commencer à réaliser des fusions à la volée, pendant qu'on calcule les SCM. Cet algorithme a été implanté peu de temps après en Java, puis dans d'autres langages de programmation. Très récemment, CPython a encore changé d'algorithme pour une nouvelle variante de TimSort que nous ne présenterons pas ici.

5.1 Fusions dans TimSort

L'algorithme de TimSort utilise une pile $\mathcal{P}$ où il stocke les séquences croissantes déjà calculées, par la position de leurs débuts et leurs longueurs. A chaque fois qu'une nouvelle SCM est ajoutée dans la pile, des fusions éventuelles ont lieu avant de calculer la SCM suivante. Si on note s_0, s_1, s_2 et s_3 les quatre séquences croissantes à partir du dessus de la pile, si elles existent, on utilise les règles suivantes (s_0 est donc la séquence en haut de la pile) :

- si $|s_2|\leq|s_1|+|s_0|$ ou $|s_3|\leq|s_2|+|s_1|$ alors on fusionne s_1 avec la plus courte des deux séquences s_0 ou s_2 et on actualise la pile ;
- sinon si $|s_1|\leq|s_0|$, on fusionne s_0 et s_1 et on actualise la pile.

Ces fusions, que l'on appellera *fusions intermédiaires* sont répétées jusqu'à ce qu'aucune ne s'applique, puis on calcule la SCM suivante, on l'ajoute dans la pile, et on ré-applique les règles de fusions intermédiaires ci-dessous, etc.

Une fois les fusions intermédiaires de la dernière SCM effectuées, il peut rester plusieurs séquences dans la pile : on les fusionne alors itérativement de haut en bas dans la pile, jusqu'à ce qu'il n'en reste qu'une, le tableau trié. C'est l'étape de *finalisation*.

La complexité de l'algorithme en $\mathcal{O}(n\log n)$ de TimSort n'a été établie qu'en 2015, elle utilise des arguments astucieux de complexité amortie. Trois ans après,

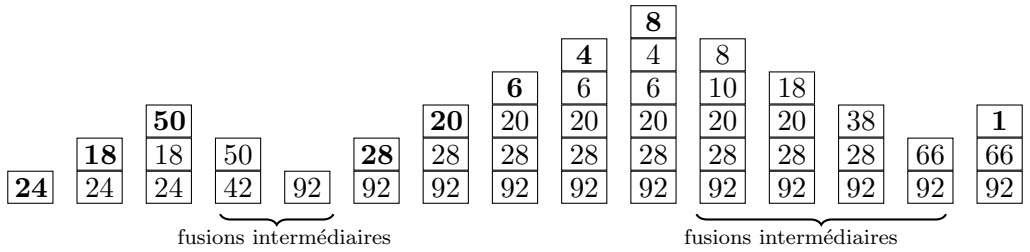

Figure 4: Exemple de l'évolution de la pile de séquences croissantes pour des SCM de longueurs $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. On a indiqué seulement la longueur des séquences dans la pile. Les SCM qui rentrent dans la pile sont indiquées en gras.

il a été démontré que TimSort est en fait en $\mathcal{O}(n\mathcal{H}(\mathcal{S}))$ et est donc optimal pour notre second cadre.

5.2 Les séquences croissantes de TimSort

L'algorithme de TimSort n'utilise pas exactement la décomposition en SCM présentée au début. Tout d'abord elle autorise la décomposition en séquences croissantes ou strictement décroissantes : à chaque fois qu'on commence une nouvelle séquence, on regarde les deux premiers éléments $T[i]$ et $T[i+1]$ (si on n'est pas au bout du tableau) et leur ordre relatif $T[i] \leq T[i+1]$ ou bien $T[i] > T[i+1]$ pour décider si on commence une séquence croissante ou strictement décroissante. Les séquences strictement décroissantes calculées sont ensuite inversées en place pour devenir croissantes. Cela permet de garantir qu'il y a au plus $\lceil n/2 \rceil$ séquences croissantes.

5.3 Bugs dans le TimSort de Java

La première version de l'algorithme de TimSort ne n'avait pas la condition $|s_3| \leq |s_2|+|s_1|$: à tout moment on ne regardait que les trois premières séquence de la pile. Dans le texte descriptif de l'algorithme de TimSort, il était annoncé qu'à chaque fois qu'on avait terminé une séquence de fusion intermédiaires (avant l'insertion d'une nouvelle SCM), on avait l'invariant : partout dans la pile $|s_{i+2}| > |s_{i+1}|+|s_i|$. Ce qui impliquerait une croissance exponentielle, de haut en bas, des longueurs des séquences de la pile à ces moments-là. Il s'avère que cette affirmation est fausse, on peut construire une séquence de SCM qui l'invalide (si on enlève la condition $|s_3| \leq |s_2| + |s_1|$).

Cette erreur ne compromettait pas la correction de l'algorithme, qui triait bien le tableau et en temps $\mathcal{O}(n \log n)$. Cependant, en Java, la pile était implantée en utilisant un tableau statique dimensionné grâce à l'invariant erroné. Il était possible de faire déborder le tableau, qui était sous-dimensionné. Comme le code

de JAVA est public, des universitaires ont pu l'analyser, trouver l'erreur et alerter la communauté de JAVA en proposant l'ajout de la condition sur s_3, ce qui garantit que l'invariant est bien respecté. Cette proposition a été suivie par JAVA et par PYTHON. La pile de PYTHON étant implantée avec un tableau redimensionnable, il ne pouvait pas y avoir de problème de dépassement de taille, mais la communauté a préféré modifier leur implantation pour que l'invariant soit correct.

Ainsi après plus de 15 ans d'existence, TIMSORT a été complètement corrigé, et au-delà de sa complexité annoncée en $\mathcal{O}(n \log n)$, il a été prouvé que l'algorithme était optimal si on prenait comme mesure de désordre l'entropie des longueurs des SCM : TIMSORT est bien $\mathcal{O}(n\mathcal{H}(\mathcal{S}) + n)$, tout en ne fusionnant que des séquences adjacentes.

6 Pistes de réflexion pour l'exposé

1. Expliciter l'algorithme de fusion de deux tableaux triés S_1 et S_2. Expliquer pourquoi, si ce sont deux sous-séquences consécutives dans un même tableau, on peut effectuer la fusion en utilisant un espace supplémentaire de seulement $\min(|S_1|, |S_2|)$, si on s'autorise à déplacer tous les éléments de S_1 et S_2 une fois supplémentaire.

2. Expliquer pourquoi la complexité du Tri Fusion Naturel est en $\mathcal{O}(n \log \rho + n)$.

3. Pour le premier cadre d'étude, on pourrait ajouter un algorithme à la partie 3.2 qui met les séquences croissantes dans une file, et qui à chaque étape prend les deux premières, les fusionne, et place le résultat dans la file. Que penser de cet algorithme ? Et si on remplace la file par une pile ?

4. Construire pour tout n assez grand une séquence de longueurs de SCM $\mathcal{S}_n = (|S_1|, \ldots, |S_{\rho_n}|)$ telle que $n \log \rho_n + n = \Theta(n \log n)$ et $n\mathcal{H}(\mathcal{S}_n) + n = \Theta(n)$. Qu'en déduire sur les algorithmes de la partie 3.2 ?

5. Expliquer pourquoi il est plus avantageux de fusionner des sous-tableaux consécutifs quand on travaille sur un grand tableau. Justifier également le choix de TIMSORT de faire des fusions "à la volée" au fur et à mesure qu'on calcule les SCM.

6. Quelle technique algorithme permet de calculer facilement la formule (5) en temps polynomial ? Quelle en serait la complexité ? On suppose qu'on a déjà une liste des SCM sous forme de paires (d, ℓ), où d est l'indice de début de la SCM et ℓ sa longueur.

7. Expliciter quelles règles ont été appliquées pour les fusions intermédiaires de l'exemple de la figure 4.

8. Que penser de l'algorithme tout simple qui calcule à la volée les SCM et les fusionne dès qu'il y en a deux ?

9. Une variante simplifiée de TimSort consiste à changer toutes les règles de fusions intermédiaires par la seule règle : si $|s_1| \leq 2|s_0|$ alors fusionner s_0 et s_1. Ecrire le pseudo-code complet de cet algorithme (en supposant les SCM déjà calculées, et la fonction de fusion disponible). Est-ce que l'invariant $|s_{i+1}| \leq |s_i|$ est respecté sur toute la pile avant chaque ajout de nouvelle SCM ?

10. On retire la condition $|s_3| \leq |s_2| + |s_1|$ des fusions intermédiaires de TimSort (c'est donc la variante historique, avant correction). Trouver une suite de longueurs de SCM telle que si l'invariant décrit dans la partie 5.3 n'est pas vérifié partout dans la pile au moment où on va effectuer l'étape de finalisation.

11. Le bug de l'implantation en Java de TimSort a pu être découvert car le code source est publiquement disponible. Cela a permis à une correction d'être très rapidement déployée. Discuter de l'impact societal et économique du choix d'avoir du code public, du code libre, ou du code privé.