L'algorithme de tri *ShiversSort* adaptatif

Vincent Jugé [1]

Nous présentons ci-dessous l'algorithme de tri ShiversSort *adaptatif. Cet algorithme est une variante du célèbre algorithme* TimSort*, qui est l'algorithme utilisé dans les bibliothèques standard de langages de programmation tels que Python et Java (pour les types non primitifs). Plus précisément, il appartient à la classe des algorithmes dits* k-aware*, classe introduite par Buss et Knop et censée caractériser les algorithmes semblables à* TimSort*.*

Nous démontrons ici que ShiversSort *adaptatif requiert un nombre de comparaisons presque optimal parmi l'ensemble des algorithmes de tri par fusion dits* naturels *: le surcoût de* ShiversSort *adaptatif est au plus linéaire en la taille du tableau à trier.* ShiversSort *adaptatif est le premier algorithme* k-aware *à jouir d'une telle propriété, qui permet entre autres une amélioration de 33 % de la complexité de* TimSort *dans le pire des cas. Tout ceci suggère que* ShiversSort *adaptatif pourrait être un substitut naturel à l'utilisation de* TimSort*.*

Introduction

Le problème du tri des données est l'un des problèmes les plus anciens et les plus étudiés dans le domaine de l'informatique. En effet, l'usage d'algorithmes de tri est

1. Maître de conférences à l'Université Gustave Eiffel, LIGM (UMR 8049), CNRS, ENPC, ESIEE Paris, UPEM, F77454, Marne-la-Vallée, France.

omniprésent, puisqu'il est souvent nécessaire en tant que brique de base de nombreux autres algorithmes. Par conséquent, dès les années 1940, de multiples algorithmes de tri ont été élaborés, chacun jouissant de diverses propriétés d'optimalité, concernant aussi bien leur complexité temporelle (et, plus précisément, en nombre de comparaisons et de déplacements d'éléments requis) que spatiale. Ainsi, chaque décennie vient avec ses nouveaux algorithmes de tri, chacun utilisant une approche différente ou consistant en l'adaptation de structures de données dédiées, et ce dans le but d'améliorer les algorithmes préexistants : on peut ainsi citer le tri fusion [8], le tri rapide [10], le tri par tas [20], ou encore les algorithmes SmoothSort [6] et SplaySort [14].

L'un de ces algorithmes est TimSort, inventé en 2002 par Tim Peters [16]. Cet algorithme a très vite été remarqué pour sa capacité à trier efficacement des données réelles, et est bientôt devenu un algorithme de tri des bibliothèques standard de langages de programmation tels que Python et Java. L'irruption d'un tel algorithme conçu de manière peu ou prou artisanale, et qui affichait cependant de meilleures performances que bien d'autres algorithmes pourtant considérés comme optimaux, a donc ravivé l'intérêt pour l'étude des algorithmes de tri.

Comprendre précisément les raisons du succès de TimSort est une tâche de longue haleine. Ces raisons sont, entre autres, le fait que TimSort soit particulièrement adapté à l'architecture des ordinateurs (par exemple dans la gestion du cache) et à des modèles de distribution de données réalistes. Un tel modèle, qui met en évidence à quel point TimSort est adapté dès qu'il s'agit de traiter des données pré-triées, se base sur la notion de *sous-suites monotones* [3, 7] (ci-après appelées runs), c'est-à-dire de sous-suites croissantes ou bien strictement décroissantes, comme illustré en Figure 1.

$$S = (\underbrace{12,7,6,5,4,3,1,0}_{1^{\text{er}}\ \textit{run}},\ \underbrace{0,7,14,36,37,42,73}_{2^{\text{e}}\ \textit{run}},\ \underbrace{3,3,5,21,21,21,24}_{3^{\text{e}}\ \textit{run}})$$

FIGURE 1. Une suite d'entiers et sa décomposition en runs : les deux premiers éléments de chaque *run* déterminent s'il s'agit d'un *run* croissant ou bien strictement décroissant.

La décomposition d'un tableau en de telles sous-suites était déjà à la base du NaturalMergeSort de Knuth [12], inventé avant TimSort, et qui consiste en la variante suivante du tri fusion classique. NaturalMergeSort procède en deux phases : dans un premier temps, et à l'aide d'un algorithme glouton, on identifie les sous-suites monotones maximales dont est formé le tableau ; puis on fusionne ces sous-suites deux à deux, à la manière du tri fusion usuel. En écho à l'apparition de cet algorithme, tout algorithme basé sur la décomposition d'un tableau en de telles sous-suites monotones est qualifié de tri fusion *naturel*.

Outre le fait qu'il s'agisse d'un tri fusion naturel, TimSort inclut également de nombreuses optimisations, soigneusement conçues et paramétrées en s'appuyant sur des jeux de tests exhaustifs. Cette méthode de conception a donné à TimSort sa structure générale, que l'on peut sommairement découper en trois parties peu ou prou indépendantes : (i) un tri par insertion un peu compliqué, utilisé pour traiter de petits *runs* (de longueur 32 ou moins); (ii) une stratégie simple pour décider quels grands *runs* (de longueur 33 ou plus) fusionner; (iii) l'usage d'une procédure assez complexe pour fusionner ces *runs*. Les première et troisième composantes sont celles qui sont les plus compliquées, et dont les paramètres ont été choisis suite à des tests empiriques. Comprendre précisément pourquoi elles sont efficaces et comment les améliorer semble donc difficile. En revanche, la deuxième composante est assez simple. C'est donc là que repose une grande partie du potentiel de modification et d'amélioration de TimSort.

Éléments de contexte.

Peu après son invention, l'algorithme TimSort a été largement adopté comme standard dans de multiples langages de programmation. La communauté scientifique a donc cherché à répliquer ce succès, en élaborant des algorithmes eux aussi très performants sur des données déjà partiellement ordonnées. Cependant, et en raison de sa conception artisanale et hors du milieu académique, TimSort s'est avéré moins facile à analyser que prévu.

Ainsi, on pourrait espérer que TimSort, comme tout algorithme de tri décent, ne nécessite que $\mathcal{O}(n\log(n))$ comparaisons pour trier des tableaux de taille n. De manière surprenante, ce résultat n'a été démontré qu'en 2015, plus d'une décennie après que TimSort avait déjà été largement déployé. Pire encore, et à cause de l'absence d'une analyse théorique et systématique de cet algorithme, certains bugs des implémentations de TimSort dans les langages Java et Python n'ont été découverts que très récemment [1, 5].

Parallèlement, et depuis l'invention de TimSort, plusieurs algorithmes de tri fusion naturels ont été élaborés, chacun étant censé jouir de bornes de complexité faciles à démontrer : ShiversSort [18], MinimalSort [3, 19], α-MergeSort [4], PowerSort [15], etc. Ces algorithmes ont souvent de nombreuses propriétés communes avec TimSort. Quelques-unes d'entre elles sont mentionnées dans la Table 1.

Par exemple, et hormis MinimalSort, tous ces algorithmes sont *stables*. Cela signifie que ces algorithmes préservent l'ordonnancement initial des éléments que l'ordre considère comme égaux. Il s'agit d'une propriété très importante des tris fusion. En effet, elle signifie que seuls des *runs* adjacents seront fusionnés, ce qui permet d'effectuer ces fusions directement sur le tableau à trier plutôt qu'en utilisant des listes chaînées. Cette propriété est également cruciale dès lors qu'il s'agit de trier des tableaux dont les éléments sont de types composites (en Java, il s'agira de tous les types non primitifs), qui pourraient être triés de deux manières différentes en fonction de diverses mesures de comparaison.

Algorithme	Complexité	Stable	*k-aware*	Coût des fusions
NaturalMergeSort	$\Theta(n+n\log(\rho))$	✓	✗	$n\log_2(\rho)+\mathcal{O}(n)$
TimSort	$\Theta(n+n\mathcal{H})$	✓	$k=4$	$3/2n\mathcal{H}+\mathcal{O}(n)$
ShiversSort	$\Theta(n\log(n))$	✓	$k=2$	$n\log_2(n)+\mathcal{O}(n)$
MinimalSort	$\Theta(n+n\mathcal{H})$	✗	✗	$n\mathcal{H}+\mathcal{O}(n)$
α-MergeSort	$\Theta(n+n\mathcal{H})$	✓	$k=3$	$c_\alpha n\mathcal{H}+\mathcal{O}(n)$
PowerSort	$\Theta(n+n\mathcal{H})$	✓	✗	$n\mathcal{H}+\mathcal{O}(n)$

TABLE 1. Propriétés de plusieurs tris fusion naturels – La constante c_α est comprise entre 1.04 et 1.09 – Le *coût des fusions* d'un algorithme est approximativement égal au nombre de comparaisons et de déplacements d'éléments que l'algorithme requiert dans le pire des cas, et donc à sa complexité en temps.

En outre, chacun de ces algorithmes trie les tableaux de taille n en temps $\mathcal{O}(n\log(n))$. En général, et dans le modèle de tri par comparaisons (où un algorithme n'a droit qu'à deux opérations de base, qui sont *comparer deux éléments* et *copier un élément à un autre endroit de la mémoire*), on ne peut pas faire mieux. Cependant, en choisissant des paramètres qui nous permettront de mieux décrire le caractère partiellement ordonné d'un tableau, on peut espérer affiner cette borne supérieure sur le temps d'exécution de l'algorithme.

Ainsi, si l'on note ρ le nombre de runs dont est formé le tableau à trier, il s'avère que tous ces algorithmes, à l'exception de ShiversSort, fonctionnent en fait en temps $\mathcal{O}(n+n\log(\rho))$. Notons ici que l'on a bien sûr $\rho \leqslant n$, et qu'il existe des tableaux de taille n tels que $\rho \geqslant n/2$. Cette nouvelle borne supérieure n'est donc pas toujours meilleure que la borne précédente, mais elle l'est *dans certains cas*, qui sont justement les cas qui nous intéressent, c'est-à-dire les cas des tableaux partiellement ordonnés.

On peut même aller plus loin dans cette démarche et choisir comme paramètres, outre la longueur du tableau et le nombre de *runs* dont il est formé, la longueur de ces *runs*, que l'on notera $r_1,\dots,r_\rho$. On peut alors démontrer que certains des algorithmes mentionnés ci-dessus fonctionnent en temps $\mathcal{O}(n+n\mathcal{H})$, où l'on a posé $\mathcal{H}=\sum_{i=1}^{\rho}(r_i/n)\log_2(n/r_i)$: remarquons ici que, si l'on note R le *run* auquel appartient un élément choisi uniformément au hasard parmi les n éléments du tableau, la quantité $\mathcal{H}$ n'est autre que l'entropie de la variable aléatoire R. On dira donc qu'il s'agit de l'*entropie* du tableau. De plus, on peut une fois encore démontrer que, dans ce nouveau cadre de travail, la borne supérieure en $\mathcal{O}(n+n\mathcal{H})$ que nous avons obtenue est en fait optimale [3, 13].

Au vu de la Table 1, quatre algorithmes ont des complexités de l'ordre de $\mathcal{O}(n+$

$n\mathscr{H}$). Afin de les distinguer, il nous faut donc mettre au point des mesures de complexité plus fines. De surcroît, et à l'exception notable de TimSort, il s'avère que tous ces algorithmes ne sont en fait caractérisés que par la stratégie utilisée pour décider quels *runs* fusionner. La sous-routine choisie pour fusionner effectivement deux *runs* y est donc implicite, et son implémentation laissée au libre choix du programmeur. Cette situation nous pousse donc à travailler dans le modèle de coût suivant.

Pour fusionner deux *runs* adjacents de longueurs m et n, un algorithme de fusion naïf requiert de l'ordre de $m+n$ comparaisons et déplacements d'éléments. Or, quel que soit l'algorithme de fusion utilisé, et dans le pire des cas, on aura toujours besoin de $m+n$ déplacements d'éléments. Par conséquent, dans tout l'article, nous mesurerons la complexité des algorithmes en terme de *coût des fusions* [2, 4, 9, 15] : le coût d'une fusion entre deux *runs* de longueurs m et n est défini comme l'entier $m+n$, et on identifie la complexité d'un algorithme à la somme des coûts des différentes fusions que l'algorithme aura exécutées pour trier un tableau.

On peut alors démontrer que le coût des fusions de n'importe quel algorithme de tri fusion naturel est d'au moins $n\mathscr{H}$, même dans le meilleur des cas. Cette remarque nous amène à identifier MinimalSort et PowerSort comme les deux seuls algorithmes dont le coût des fusions est optimal. Et, puisque PowerSort est stable, cet algorithme est donc un candidat naturel pour succéder à TimSort en tant qu'algorithme de tri standard dans les langages Python et Java. Néanmoins, et même s'il admet des implémentations similaires à celles de TimSort, la stratégie qu'il utilise pour décider quels *runs* fusionner est nettement plus complexe.

La question de l'existence d'un algorithme de tri fusion naturel simple dont la structure semblable à celle de TimSort et dont le coût des fusions serait optimal (à un terme additif linéaire en n près) reste donc ouverte.

Une première étape, sur le chemin de cet objectif, consiste à définir une notion adéquate permettant de caractériser ce qui fait qu'un algorithme est simple et semblable à TimSort. La stratégie de TimSort consiste à découvrir les *runs* à la volée, à la manière d'un algorithme glouton, et à les stocker dans une pile : si un *run* couvre les éléments en positions $i, i+1, \ldots, j$ dans le tableau, la pile contiendra la paire (i, j). Puis TimSort ne procède qu'à des fusions entre *runs* stockés dans le dessus de la pile, et les décisions qu'il prend ne sont basées que sur les longueurs de ces *runs* situés au-dessus de la pile. Procéder ainsi s'avère très avantageux au vu de l'architecture des ordinateurs actuels, par exemple parce que cela nous permet de tirer largement parti de l'existence de mémoire cache dans les processeurs.

C'est sur la base de ce constat qu'a été inventée la notion d'algorithme *k-aware* [4]. Un tri fusion naturel est *k-aware* si les *runs* qu'il fusionne figurent tous parmi les k premiers *runs* de la pile, et si les décisions qu'il prend ne sont basées que sur les longueurs de ces k premiers *runs*. La 4e colonne de la Table 1 indique quels algorithmes sont *k-aware* pour un paramètre k fini, auquel cas elle indique également la plus petite valeur de k possible.

Se concentrer sur la recherche d'algorithmes *k-aware* nous permet donc de nous concentrer sur la partie (ii) de TimSort, sans modifier les composantes (i) et (iii) qui, si elles ont elles aussi été mises au point en dehors d'un cadre académique, ont néanmoins été l'objet de moult optimisations. Par conséquent, si l'on amende de manière marginale la stratégie de fusion de TimSort, on peut raisonnablement espérer que toutes les optimisations effectuées sur les composantes (i) et (iii) de TimSort resteront valides, et ce même si elles ne sont pas comprises dans le détail. C'est pourquoi, dans la suite, nous identifierons chaque algorithme de tri fusion naturel avec sa stratégie de fusion des *runs*, quitte à intégrer les composantes (i) et (iii) de TimSort ultérieurement.

Contributions.

Nous présentons un nouvel algorithme de tri fusion, appelé ShiversSort adaptatif. Comme mentionné précédemment, cet algorithme sera assimilé à la stratégie qu'il suit pour savoir quels *runs* fusionner. ShiversSort adaptatif se situe à mi-chemin entre les algorithmes TimSort et ShiversSort, avec pour objectif de bénéficier des forces de chacun de ces deux algorithmes. Il en résulte un algorithme dont la stratégie est extrêmement similaire à celle de TimSort, au point que passer de l'un à l'autre ne nécessiterait de modifier qu'une dizaine de lignes (sur un millier) du code de TimSort en Java.

ShiversSort adaptatif est un algorithme 3-*aware* et stable, dont le coût des fusions est égal à $n\mathscr{H} + \mathscr{O}(n)$. ShiversSort adaptatif apparaît donc comme optimal vis-à-vis de tous les critères mentionnés dans la Table 1. En particulier, il s'agit du premier algorithme *k-aware* doté d'un coût des fusions en $n\mathscr{H} + \mathscr{O}(n)$, ce qui nous permet de répondre positivement à une conjecture de Buss et Knop [4]. De plus, sa stratégie simple nous permet d'obtenir une preuve de complexité simple elle aussi ; nous en proposons, ci-dessous, une version courte mais complète.

ShiversSort adaptatif et algorithmes apparentés

Nous décrivons, dans cette section, la stratégie qu'utilisent ShiversSort adaptatif et plusieurs autres algorithmes de tri fusion naturels pour décider quels *runs* fusionner. La stratégie de ShiversSort adaptatif est présentée dans l'algorithme 1.

Comme TimSort, cet algorithme consiste à découvrir des *runs*, c'est-à-dire des sous-suites monotones, et à les maintenir dans une pile. Au cours de l'algorithme, et selon que l'on tombe dans un des cas n° 1 à 4, on pourra soit fusionner deux des *runs* situés en haut de la pile, soit insérer un nouveau *run* dans la pile. En particulier, puisque les conditions régissant les cas n° 1 à 4 ne dépendent que des valeurs des entiers ℓ_1, ℓ_2 et ℓ_3, et puisque seuls les *runs* R_1, R_2 et R_3 peuvent être fusionnés, l'algorithme ShiversSort adaptatif appartient à la famille des algorithmes 3-*aware* définie par Buss et Knop [4].

Algorithm 1: ShiversSort adaptatif

Entrée : Tableau T à trier

Résultat : Le tableau T est trié. Il consiste en un unique *run*, qui reste stocké dans la pile $\mathscr{P}$.

Remarque: On note h la hauteur de la pile $\mathscr{P}$, et R_i son i-ième plus haut élément (avec $1 \leqslant i \leqslant h$). On note r_i la longueur du *run* R_i, et on pose $\ell_i = \lfloor \log_2(r_i) \rfloor$. Dans la pile $\mathscr{P}$, chaque run est représenté par deux pointeurs vers ses premier et dernier éléments. Quand on fusionne deux *runs* consécutifs, on les remplace, dans $\mathscr{P}$, par le *run* résultant de la fusion.

```
runs ← décomposition de T en runs
𝒫 ← pile vide
tant que vrai :                                          ▷ boucle principale
  si h ⩾ 3 et ℓ1 ⩾ ℓ3          : fusionner les runs R2 et R3        ▷ cas n° 1
  sinon, si h ⩾ 3 et ℓ2 ⩾ ℓ3   : fusionner les runs R2 et R3        ▷ cas n° 2
  sinon, si h ⩾ 2 et ℓ1 ⩾ ℓ2   : fusionner les runs R1 et R2        ▷ cas n° 3
  sinon, si runs ≠ ∅           : ôter un run R de runs et
                                 l'insérer dans la pile 𝒫          ▷ cas n° 4
  sinon                        : sortir de la boucle principale
tant que h ⩾ 2 :
  fusionner les runs R1 et R2
```

Le lecteur aura pu noter que les cas n° 1 et n° 2 ont tous deux pour conséquence la fusion entre les *runs* R_2 et R_3. Nous avons décidé de séparer ces deux cas afin de simplifier l'analyse de complexité effectuée c.

Présentons maintenant quelques-uns des algorithmes *k-aware* mentionnés dans la Table 1. Bien qu'apparentés à ShiversSort adaptatif, ils utilisent une stratégie différente, que l'on peut en fait obtenir en ne modifiant que la boucle principale de ShiversSort adaptatif.

Voici tout d'abord l'algorithme TimSort, conçu par Tim Peters [16], et dont la boucle principale est présentée ci-dessous :

Comme mentionné dans l'introduction, le coût des fusions de TimSort est en $\mathscr{O}(n + n\mathscr{H})$. Ce résultat repose sur les deux ingrédients suivants : (i) on s'abstient de fusionner un *run* nouvellement arrivé au sommet de la pile tant qu'il n'est pas de taille comparable avec celle des autres *runs* situés en haut de la pile, et (ii) on maintient un invariant (l'inégalité $r_i + r_{i+1} \leqslant r_{i+2}$ pour tout $i \geqslant 3$) grâce auquel les tailles des *runs*, quand on s'enfonce dans les profondeurs de la pile, croissent à vitesse exponentielle.

tant que `vrai` **:** ▷ boucle principale
- **si** $h \geqslant 3$ et $r_1 > r_3$ **:** fusionner les *runs* R_2 et R_3 ▷ cas ♢
- **sinon, si** $h \geqslant 2$ et $r_1 \geqslant r_2$ **:** fusionner les *runs* R_1 et R_2 ▷ cas ♠
- **sinon, si** $h \geqslant 3$ et $r_1 + r_2 \geqslant r_3$ **:** fusionner les *runs* R_1 et R_2 ▷ cas ♣
- **sinon, si** $h \geqslant 4$ et $r_2 + r_3 \geqslant r_4$ **:** fusionner les *runs* R_1 et R_2 ▷ cas ♡
- **sinon, si** `runs` $\neq \varnothing$ **:** ôter un *run* R de `runs` et l'insérer dans la pile $\mathscr{P}$ ▷ cas n° 4
- **sinon** **:** sortir de la boucle principale

Notons au passage que le cas ♡ avait été omis dans la version originale de TimSort, et que cette omission était passée inaperçue en raison de l'absence d'analyse de complexité de l'algorithme. Cependant, et en l'absence du cas ♡, l'invariant que TimSort était censé maintenir n'était en fait plus nécessairement vérifié, et la complexité en mémoire de l'algorithme était supérieure à ce qui avait été prévu. Il en a résulté l'apparition de multiples bugs, liés à des dépassements de mémoire, dans les implémentations de TimSort en Python et en Java [1, 5].

Par ailleurs, une analyse fine de la complexité de TimSort, dans sa version actuelle, permet de démontrer que la constante multiplicative cachée dans la notation $\mathscr{O}$ est assez élevée, puisqu'elle est de 50% supérieure à la constante optimale. C'est l'objet du résultat ci-dessous, démontré dans [1, 4].

Théorème 1. *Le coût des fusions de TimSort, sur des tableaux de longueur n et d'entropie $\mathscr{H}$, est inférieur ou égal à $3/2\,n\mathscr{H} + \mathscr{O}(n)$. En outre, il existe des tableaux de longueur n pour lesquels le coût des fusions de TimSort est de l'ordre de $3/2\,n\log_2(n) + \mathscr{O}(n)$.*

Puisque $\mathscr{H} \leqslant \log_2(n)$, ces deux bornes coïncident bien l'une avec l'autre. Ainsi, le coût des fusions de TimSort peut effectivement être de l'ordre de $3/2\,n\mathscr{H} + \mathscr{O}(n)$, et ce même dans le cas le plus défavorable, où l'on a déjà $\mathscr{H} = \log_2(n) + \mathscr{O}(1)$.

Afin de faire baisser cette constante de $3/2$ à 1, il était donc important de s'intéresser à d'autres stratégies de fusion. Dès lors que l'on a un tel objectif en tête, un premier espoir vient de l'algorithme ShiversSort, inventé par Olin Shivers [18]. On peut obtenir cet algorithme en omettant les cas n° 1 et 2 de ShiversSort adaptatif, c'est-à-dire en utilisant la boucle principale suivante :

tant que `vrai` **:** ▷ boucle principale
- **si** $h \geqslant 2$ et $\ell_1 \geqslant \ell_2$ **:** fusionner les *runs* R_1 et R_2 ▷ cas n° 3
- **sinon, si** `runs` $\neq \varnothing$ **:** ôter un *run* R de `runs` et l'insérer dans la pile $\mathscr{P}$ ▷ cas n° 4
- **sinon** **:** sortir de la boucle principale

Contrairement à TimSort et à ShiversSort adaptatif, ShiversSort peut fusionner un *run* R dès son insertion dans la pile, et ce y compris si R est beaucoup plus long que les autres *runs* de la pile. Par conséquent, cet algorithme ne s'adapte pas au nombre de *runs*, ni à leurs longueurs. Il jouit néanmoins d'une propriété remarquable, puisque son coût des fusions s'avère optimal (à un terme additif linéaire en n près), dans le pire des cas, si l'on se restreint à choisir n comme seul paramètre. C'est l'objet du résultat suivant, démontré dans [4, 18].

Théorème 2. *Le coût des fusions de ShiversSort, sur des tableaux de longueur n, est inférieur ou égal à* $n\log_2(n) + \mathcal{O}(n)$. *En outre, il existe des tableaux de longueur n et qui se décomposent en* ρ *sous-suites monotones pour lesquels le coût des fusions de ShiversSort est de l'ordre de* $\omega(n\log_2(\rho))$.

La preuve de ce résultat, que nous éludons ici, est très semblable à notre propre analyse de la complexité de ShiversSort adaptatif, menée ci-après. Un élément clé des deux démonstrations, et qui sera énoncé dans le Lemme 5, est le fait que la suite $\ell_1, \ell_2, \ldots, \ell_h$ soit strictement croissante à partir d'un certain rang : ce rang est 2 dans la preuve de [4], et il vaut 3 dans le Lemme 5. Cet invariant est remarquablement similaire à celui de TimSort, en ce qu'il impose aux longueurs des *runs* stockés dans la pile de croître à vitesse exponentielle. Cependant, c'est ce nouvel invariant qui nous permet de faire décroître la constante cachée dans la notation $\mathcal{O}$ de $3/2$ à 1.

Mentionnons enfin l'algorithme PowerSort, déjà cité dans l'introduction, et qui jouit d'excellentes garanties sur son temps d'exécution, puisque son coût des fusions est majoré par $n(\mathcal{H} + 2)$. Toutefois, il ne s'agit pas d'un algorithme *k-aware* pour quelque entier k que ce soit. En effet, la stratégie qu'utilise cet algorithme ne s'appuie pas uniquement sur les longueurs des *runs* stockés dans la pile, mais également sur leur positionnement au sein du tableau à trier, et cette stratégie s'avère donc plus compliquée que celle de TimSort.

Coût des fusions de ShiversSort adaptatif

Nous avons affirmé, dans l'introduction, que le coût des fusions de ShiversSort adaptatif était excellent, puisqu'il était égal à $n\mathcal{H} + \mathcal{O}(n)$. C'est ce que nous allons démontrer dans cette section, où nous effectuons une première analyse de complexité de ShiversSort adaptatif. Le résultat central est donc le Théorème 3 ci-dessous, qui nous assure que le coût des fusions de ShiversSort adaptatif est de l'ordre de $n\mathcal{H} + \mathcal{O}(n)$.

Théorème 3. *Le coût des fusions de ShiversSort adaptatif, sur les tableaux de taille n et d'entropie* $\mathcal{H}$, *est inférieur ou égal à* $n(\mathcal{H} + 19)$.

Remarque. Comme nous pourrons l'illustrera le Théorème 12, la constante 19 est grossièrement surévaluée. Cependant, elle est plus facile à obtenir que d'autres constantes plus fines, et c'est pourquoi nous nous en contenterons pour l'instant.

Dans la suite, nous noterons r la longueur d'un *run* R, et ℓ l'entier $\lfloor \log_2(r) \rfloor$. On dira que ℓ est le *niveau* du *run* R. Nous adapterons librement ces notations en fonction du nom des runs considérés. Ainsi, nous utiliserons sans autre forme d'avertissement les notations r' et ℓ' pour désigner la longueur d'un *run* R' et l'entier $\lfloor \log_2(r') \rfloor$. De surcroît, il nous arrivera fréquemment de nous intéresser aux *runs* contenus dans la pile à un moment donné de l'exécution de l'algorithme. On notera alors $(R_1, \ldots, R_h)$ cette pile et h sa hauteur, le *run* R_k étant situé en $k^{\text{ème}}$ position en partant du haut. Puis, comme précédemment, on notera r_k et ℓ_k la longueur de R_k et l'entier $\lfloor \log_2(r_k) \rfloor$.

Forts de ces quelques notations, nous pouvons maintenant démontrer deux lemmes concernant les niveaux des *runs* manipulés lors d'une exécution de l'algorithme.

Lemme 4. *Lorsque deux* runs *R et R' sont fusionnés en un unique* run *R'', on a* $\ell' \leqslant \max\{\ell, \ell'\} + 1$.

Démonstration. Sans perte de généralité, on peut supposer que $r \leqslant r'$. On constate alors que

$$2^{\ell''} \leqslant r'' = r + r' \leqslant 2r' < 2 \times 2^{\ell'+1} \leqslant 2^{\ell'+2},$$

et donc que $\ell'' \leqslant \ell' + 1$. □

Lemme 5. *Soit $\mathscr{P} = (R_1, \ldots, R_h)$ la pile obtenue à un moment quelconque de l'exécution de la boucle principale de* ShiversSort *adaptatif. Alors*

$$\ell_2 \leqslant \ell_3 < \ell_4 < \ldots < \ell_h. \tag{1}$$

Démonstration. Nous allons procéder par récurrence sur le nombre d'étapes (insertions d'un *run* dans la pile ou fusions de deux *runs*) déjà effectuées par l'algorithme. Tout d'abord, si $h \leqslant 2$, il n'y a rien à démontrer ; cette situation se produit entre autres au début de l'algorithme.

Démontrons maintenant que, si une pile $\mathscr{P} = (R_1, \ldots, R_h)$ satisfait (1) et est transformée en une pile $\overline{\mathscr{P}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ par une fusion des *runs* R_1 et R_2, ou bien R_2 et R_3, ou bien par l'insertion d'un nouveau *run* $\overline{R}_1$, la nouvelle pile $\overline{\mathscr{P}}$ satisfait elle aussi (1). Pour ce faire, procédons par disjonction de cas :

— Si l'on vient de fusionner deux *runs*, alors on sait que $\overline{h} = h - 1$, que $\overline{R}_i = R_{i+1}$ pour tout $i \geqslant 3$, et que le *run* $\overline{R}_2$ est égal à R_3 (si l'on vient de fusionner R_1 et R_2) ou bien au produit de la fusion des *runs* R_2 et R_3 (si ce sont ces deux *runs* que l'on vient de fusionner). En vertu du Lemme 4, il s'ensuit que $\overline{\ell}_2 = \ell_3$ ou bien que $\overline{\ell}_2 \leqslant \max\{\ell_2, \ell_3\} + 1 = \ell_3 + 1$; dans tous les cas, on a bien $\overline{\ell}_2 \leqslant \ell_3 + 1$. Par ailleurs, puisque la pile $\mathscr{P}$ satisfait (1) on sait déjà que $\ell_3 < \ell_4 = \overline{\ell}_3 < \overline{\ell}_4 < \ldots < \overline{\ell}_{\overline{h}}$. On en déduit que $\overline{\ell}_2 \leqslant \ell_3 + 1 \leqslant \overline{\ell}_3$, de sorte que la pile $\overline{\mathscr{P}}$ satisfait elle aussi (1).

— Si l'on vient d'insérer le *run* $\overline{R}_1$ dans la pile, alors on sait que $\overline{h} = h+1$ et que $\overline{R}_i = R_{i-1}$ pour tout $i \geqslant 2$. Puisque la pile $\mathscr{P}$ satisfait (1), on en déduit donc déjà que $\overline{\ell}_4 < \overline{\ell}_5 < \ldots < \overline{\ell}_{\overline{h}}$. En outre, puisque l'on vient de déclencher le cas n° 4, la pile $\mathscr{P}$ ne satisfait aucune des conditions pour déclencher les cas n° 1 à 3. Cela signifie que $\ell_1 < \ell_2 < \ell_3$ ou encore que $\overline{\ell}_2 < \overline{\ell}_3 < \overline{\ell}_4$. On en conclut que la pile $\overline{\mathscr{P}}$ satisfait elle aussi (1).

□

Le Lemme 5 stipule que les longueurs des *runs* stockés dans la pile augmentent à vitesse exponentielle (au fur et à mesure que l'on s'éloigne du sommet de la pile) ; la seule exception éventuelle était le *run* situé au sommet de la pile, et sur la longueur duquel nous n'avons aucun contrôle. Comme annoncé en pages 12 et 13, ce genre de propriété s'est avéré fondamental dans les analyses de complexité d'algorithmes tels que ShiversSort, TimSort et α-MergeSort.

La démonstration du Théorème 3 consiste dès lors en une évaluation soigneuse du coût des fusions qu'exécute l'algorithme. À cette fin, nous allons distinguer trois types de fusions : les fusions *équilibrées*, entre deux *runs* R et R' tels que $\ell = \ell'$; les fusions *déséquilibrées*, entre deux *runs* R et R' tels que $\ell \neq \ell'$ et survenues dans la boucle principale ; enfin, les fusions *tardives*, entre deux *runs* R et R' tels que $\ell \neq \ell'$ et survenues en dernière ligne de l'algorithme.

Dans la suite, le coût de chaque fusion équilibrée entre deux *runs* R et R' sera alloué équitablement aux éléments des deux *runs* concernés : chaque élément paiera un coût de 1, ce qui permettra bien de couvrir le coût total de la fusion, égal à $r+r'$. Au contraire, le coût de chaque fusion déséquilibrée sera intégralement alloué aux éléments du dernier *run* $R^\bullet$ à avoir été inséré dans la pile : chacun de ses $r^\bullet$ éléments devra payer un coût de $(r+r')/r^\bullet$. Enfin, nous verrons comment traiter à peu de frais le cas des fusions tardives.

Munis de cette politique d'allocation des coûts, nous pouvons dès à présent évaluer le coût des fusions équilibrées.

Lemme 6. *Le coût total des fusions équilibrées est inférieur ou égal à* $n(\mathscr{H}+1)$.

Démonstration. Lors d'une exécution de l'algorithme, les éléments d'un *run* R de taille initiale r peuvent prendre part à un maximum de

$$\lfloor \log_2(n) \rfloor - \ell = \lfloor \log_2(n) \rfloor - \lfloor \log_2(r) \rfloor \leqslant \log_2(n) - \log_2(r) + 1 = \log_2(n/r) + 1$$

fusions équilibrées : chaque élément paiera donc au plus $\log_2(n/r)+1$ au titre de ces fusions équilibrées. Par conséquent, si le tableau se décompose initialement en des *runs* de longueurs $r_1, r_2, \ldots, r_\rho$, la somme des coûts des fusions équilibrées est inférieure ou égale à $\sum_{i=1}^{\rho} r_i(\log_2(n/r_i)+1) = n(\mathscr{H}+1)$. □

Nous allons maintenant évaluer le coût des fusions déséquilibrées. Dans ce but, distinguons également deux types de piles : on dira qu'une pile $\mathscr{P} = (R_1, \ldots, R_h)$ est *stable* si $h \leqslant 1$ ou si $h \geqslant 2$ et $\ell_1 \leqslant \ell_2$, et qu'elle est *instable* si $h \geqslant 2$ et $\ell_1 > \ell_2$. Les piles stables jouissent de remarquables propriétés de stabilité (d'où leur nom) et sont faciles à obtenir, comment le soulignent les deux résultats suivants.

Lemme 7. *Soit* **F** *une opération de fusion exécutée durant la boucle principale de* ShiversSort *adaptatif. Soit* $\mathscr{P}$ *et* $\overline{\mathscr{P}}$ *les piles obtenues juste avant que* **F** *ne soit exécutée et juste après que* **F** *a été exécutée. Si* $\mathscr{P}$ *est une pile stable, alors* **F** *est une fusion équilibrée, et* $\overline{\mathscr{P}}$ *est stable également.*

Démonstration. Soit $(R_1, \ldots, R_h)$ et $(\overline{R}_1, \ldots, \overline{R}_{h-1})$ les *runs* respectivement contenus dans les piles $\mathscr{P}$ et $\overline{\mathscr{P}}$. Sans perte de généralité, et quitte à ajouter tout en bas de la pile $\mathscr{P}$ un *run* R_{h+1} de longueur $2(r_1 + \ldots + r_h)$, qui n'influera ni sur le caractère stable de $\mathscr{P}$, ni sur la fusion **F** et le cas qui l'a provoquée, ni sur le fait que (1) soit vraie, on suppose que $h \geqslant 3$.

Puisque $\mathscr{P}$ est stable, (1) nous assure déjà que $\ell_1 \leqslant \ell_2 \leqslant \ell_3$. On procède alors par disjonction de cas, en fonction du cas qui a provoqué la fusion **F**.

— Si **F** a été provoquée par l'un des cas n° 1 ou 2, c'est que $\ell_1 \geqslant \ell_3$ ou $\ell_2 \geqslant \ell_3$, donc que $\ell_1 \leqslant \ell_2 = \ell_3$. Comme **F** est une fusion entre les *runs* R_2 et R_3, cette fusion est donc équilibrée. Elle produit un *run* $\overline{R}_2$ tel que $\overline{\ell}_2 \geqslant \ell_2$ et, puisque $\overline{R}_1 = R_1$, on en conclut que $\overline{\ell}_1 = \ell_1 \leqslant \ell_2 \leqslant \overline{\ell}_2$. Cela signifie bien que $\overline{\mathscr{P}}$ est une pile stable.

— Si **F** a été provoquée par le cas n° 3, c'est que $\ell_3 > \ell_1 \geqslant \ell_2$, ce sans quoi on aurait déjà déclenché le cas n° 1, et donc que $\ell_1 = \ell_2 < \ell_3$. Comme **F** est une fusion entre les *runs* R_1 et R_2, cette fusion est donc équilibrée. En outre, en vertu du Lemme 4, elle produit un *run* $\overline{R}_1$ tel que $\overline{\ell}_1 \leqslant \ell_1 + 1$. Puisque $\ell_1 < \ell_3$ et que $\overline{R}_2 = R_3$, il s'ensuit que $\overline{\ell}_1 \leqslant \ell_3 = \overline{\ell}_2$. Cela signifie ici aussi que $\overline{\mathscr{P}}$ est une pile stable.

□

Lemme 8. *Soit* **F** *une opération de fusion exécutée durant la boucle principale de* ShiversSort *adaptatif, et soit* $\overline{\mathscr{P}}$ *la pile obtenue juste après que* **F** *a été exécutée. Si* **F** *a été provoquée par l'un des cas n° 2 ou 3, alors* $\overline{\mathscr{P}}$ *est une pile stable.*

Démonstration. Soit $\mathscr{P}$ la pile obtenue juste avant que **F** ne soit exécutée, et soit $(R_1, \ldots, R_h)$ et $(\overline{R}_1, \ldots, \overline{R}_{h-1})$ les *runs* respectivement contenus dans les piles $\mathscr{P}$ et $\overline{\mathscr{P}}$. Comme dans la démonstration du Lemme 7, et quitte à ajouter tout en bas de la pile $\mathscr{P}$ un *run* R_{h+1} de longueur $2(r_1 + \ldots + r_h)$, on suppose que $h \geqslant 3$.

Tout d'abord, si $\mathscr{P}$ est stable, le Lemme 7 nous assure déjà que $\overline{\mathscr{P}}$ est stable elle aussi. On suppose donc ci-dessous que $\mathscr{P}$ est instable et que **F** n'a pas été provoquée par le cas n° 1, ce qui signifie que $\ell_2 < \ell_1 < \ell_3$. Dans ces conditions, la fusion **F** n'a pu être provoquée que par le cas n° 3. Ainsi, $\overline{R}_1$ est le produit de la fusion entre les

deux *runs* R_1 et R_2, et le Lemme 7 nous assure que $\overline{\ell}_1 \leqslant \ell_1 + 1$. Puisque $\overline{R}_2 = R_3$, on en conclut que $\overline{\ell}_1 \leqslant \ell_3 = \overline{\ell}_2$, ce qui signifie bien que $\overline{\mathscr{P}}$ est une pile stable. □

Ces deux résultats nous permettent alors d'obtenir l'estimation suivante sur le coût que devront payer les éléments d'un *run* au titre des fusions déséquilibrées, et dont le Corollaire 10 découle immédiatement.

Lemme 9. *Soit R un* run *initial du tableau que* ShiversSort *adaptatif est en train de trier. Le coût total des fusions déséquilibrées pour lesquelles devront payer les éléments de R est inférieur ou égal à* $10r$.

Démonstration. Soit $\mathscr{P} = (R_1, \ldots, R_h)$ la pile obtenue juste avant que le *run* R n'y soit inséré, et soit $s = r + r_1 + \ldots + r_h$. Comme dans les démonstrations des lemmes précédents, et quitte à ajouter tout en bas de la pile $\mathscr{P}$ des *runs* R_{h+1}, R_{h+2} et R_{h+3} de longueurs respectives $2s$, $4s$ et $8s$, qui n'auront aucune influence sur la boucle principale de l'algorithme, on suppose que $h \geqslant 3$ et que $\ell_h > \ell$.

Puisque l'insertion du *run* R a été provoquée par le cas n° 4, on sait que $\ell_1 < \ell_2 < \ell_3$, ce sans quoi on aurait déclenché l'un des cas n° 1 à 3. On déduit donc de (1) que $\ell_1 < \ldots < \ell_h$. Dans ces conditions, soit k le plus petit entier tel que $\ell_{k+1} > \ell$; puisque l'on a supposé que $\ell_h > \ell$, un tel entier k existe bien.

Tout d'abord, si $\ell < \ell_1$, les éléments du *run* R ne devront payer le coût d'aucune fusion déséquilibrée, et le résultat du lemme 9 est déjà acquis. On suppose donc ci-dessous que $\ell \geqslant \ell_1$, de sorte que $k \geqslant 1$ et que $\ell_k \leqslant \ell$.

L'algorithme ShiversSort adaptatif nous impose alors de commencer par $k-1$ fusions, toutes provoquées par le cas n° 1, entre les *runs* R_1 et R_2, puis entre le *run* issu de cette fusion et R_3, et ainsi de suite jusqu'à fusionner R_k. Le coût total de toutes ces fusions est égal à $\mathsf{C} = -r_1 + \sum_{i=1}^{k}(k+1-i)r_i$, et il en résulte un *run* $\overline{R}_2$ de longueur $\overline{r}_2 = \sum_{i=1}^{k} r_i$.

Une fois que toutes ces fusions ont été exécutées, on obtient une pile $\overline{\mathscr{P}} = (\overline{R}_1, \overline{R}_2, \ldots, \overline{R}_{h+1-k})$, avec $\overline{R}_1 = R$ et $\overline{R}_i = R_{i+k-1}$ pour tout $i \geqslant 3$; le *run* $\overline{R}_2$ est le *run* issu de toutes les fusions précédentes. On constate alors, puisque $\overline{\mathscr{P}}$ vérifie (1), que $\overline{\ell}_1 < \overline{\ell}_3 < \overline{\ell}_4 < \ldots < \overline{\ell}_h$.

On distingue alors deux cas, selon que la pile $\overline{\mathscr{P}}$ est stable ou non.

— Si $\overline{\mathscr{P}}$ est stable, alors le Lemme 7 indique que toutes les fusions qui suivront, et ce jusqu'à ce que l'on insère un nouveau *run* dans la pile ou que l'on sorte de la boucle principale, seront des fusions équilibrées. Dans ce cas, les éléments de R n'auront donc plus à payer de fusion déséquilibrée.

— Si $\overline{\mathscr{P}}$ est instable, alors $\overline{\ell}_2 < \overline{\ell}_1 < \overline{\ell}_3$, donc l'algorithme déclenche le cas n° 3 et fusionne les *runs* $\overline{R}_1$ et $\overline{R}_2$, pour un coût de $\overline{r}_1 + \overline{r}_2 \leqslant 2\overline{r}_1 = 2r$. Le Lemme 8 nous assure que la pile qui résulte de cette fusion est stable, et le même raisonnement que ci-dessus nous montre que les éléments de R n'auront plus à payer de fusion déséquilibrée ultérieure.

Par conséquent, les seules fusions déséquilibrées que doivent payer les éléments de R sont certaines des $k-1$ fusions initiales, provoquées par le cas n° 1, ainsi que l'éventuelle fusion entre $\overline{R}_1$ et $\overline{R}_2$ qui aura pu s'ensuivre. Le coût total de ces fusions déséquilibrées est donc inférieur ou égal à $\mathsf{C}' = \mathsf{C} + 2r$.

Or, puisque l'on a vu que $\ell_1 < \ldots < \ell_h$ et que $\ell_k \leqslant \ell$, on sait en fait que $r_i < 2^{\ell_i+1} \leqslant 2^{\ell+i+1-k} \leqslant 2^{i+1-k} r$ pour tout $i \leqslant k$. On en déduit que

$$\mathsf{C} \leqslant \sum_{i=1}^{k} (k+1-i) r_i \leqslant \sum_{i=1}^{k} (k+1-i) 2^{i+1-k} r \leqslant \sum_{i=-\infty}^{k} (k+1-i) 2^{i+1-k} r = 8r,$$

de sorte que $\mathsf{C}' = \mathsf{C} + r + \overline{r}_2 \leqslant 10r$. □

Corollaire 10. *Le coût total des fusions déséquilibrées est inférieur ou égal à* $10n$.

Il nous reste, enfin, à évaluer le coût des fusions tardives, ce que l'on fait on suivant le modèle que nous a fourni la démonstration précédente.

Lemme 11. *Le coût total des fusions tardives est inférieur ou égal à* $8n$.

Démonstration. Soit $\mathscr{P} = (R_1, \ldots, R_h)$ la pile obtenue au moment où se termine la boucle principale de ShiversSort adaptatif. On sait que $\ell_1 < \ell_2 < \ell_3$, ce sans quoi on aurait déclenché l'un des cas n° 1 à 3 au lieu de sortir de la boucle principale. On déduit donc de (1) que $\ell_1 < \ldots < \ell_h$.

L'algorithme ShiversSort adaptatif nous impose alors de procéder à $h-1$ fusions, entre les *runs* R_1 et R_2, puis entre le *run* issu de cette fusion et R_3, et ainsi de suite jusqu'à fusionner R_h. Le coût total de toutes ces fusions est égal à $\mathsf{C} = -r_1 + \sum_{i=1}^{h} (h+1-i) r_i$; ce coût inclut la somme des coûts des fusions tardives aussi bien que d'éventuelles fusions équilibrées.

Or, on sait que $r_i < 2^{\ell_i+1} \leqslant 2^{\ell_h+i+1-h} \leqslant 2^{i+1-h} r_h \leqslant 2^{i+1-h} n$ pour tout $i \leqslant h$. On en déduit que

$$\mathsf{C} \leqslant \sum_{i=1}^{h} (h+1-i) r_i \leqslant \sum_{i=1}^{h} (h+1-i) 2^{i+1-h} n \leqslant \sum_{i=-\infty}^{h} (h+1-i) 2^{i+1-h} n = 8n,$$

ce qui conclut. □

Compléments et détails d'implémentation

Dans cette section conclusive, nous donnons quelques détails permettant, selon les cas, d'améliorer les résultats démontrés précédemment ou bien de simplifier l'implémentation que l'on ferait de ShiversSort adaptatif. Pour des raisons de concision, les résultats mentionnés ci-dessous n'ont pas vocation à être démontrés ici. Le lecteur curieux pourra se référer à la version complète de cet article (en anglais) [11].

Une meilleure borne supérieure sur le coût des fusions.

Nous avons formulé en page 14 la remarque suivante : le Théorème 3 nous indique certes que ShiversSort adaptatif jouit d'un coût des fusions de l'ordre de $n\mathscr{H}+\mathscr{O}(n)$, mais la constante cachée dans la notation $\mathscr{O}$ est assez élevée. Cependant, on peut en fait améliorer grandement cette constante. C'est l'objet du résultat suivant.

Théorème 12. *Le coût des fusions de* ShiversSort *adaptatif, sur les tableaux de taille n et d'entropie $\mathscr{H}$, est inférieur ou égal à $n\,(\mathscr{H}+\Delta)$, où $\Delta = 24/5-\log_2(5) \approx$* 2.478.

On peut en outre démontrer que la constante Δ mentionnée ici est optimale, comme en témoigne l'exemple ci-dessous.

Exemple. Soit $k \geqslant 3$ un entier, et soit $m = 2^k$. Considérons un tableau T de longueur $n = 5m$, et qui se décompose en sept *runs* de longueurs respectives $r_1 = 2$, $r_2 = 2m-10$, $r_3 = 2$, $r_4 = m+1$, $r_5 = 2$, $r_6 = 2m+2$ et $r_7 = 1$. Alors ce tableau est d'entropie $\mathscr{H} = \log_2(5)-4/5+o(1)$ et le coût des fusions de ShiversSort adaptatif sur le tableau T est égal à $\mathsf{c} = 20m-23 = n\,(\mathscr{H}+\Delta+o(1))$.

Une borne inférieure sur le coût des fusions.

Nous avons mentionné, dans l'introduction, que le coût des fusions d'un algorithme de tri fusion naturel était d'au moins $n\mathscr{H}$, même dans le meilleur des cas. Ce résultat, qui est en fait une conséquence du premier théorème de Shannon [17], peut également être démontré directement, par exemple au moyen du lemme suivant, et en se rappelant qu'un tableau entièrement trié est d'entropie nulle.

Lemme 13. *Soit T un tableau de longueur n et d'entropie $\mathscr{H}$, formé de ρ runs $R_1,\ldots,R_\rho$, avec $\rho \geqslant 2$. Puis, étant donné un entier $k \leqslant \rho-1$, soit $\overline{T}$ le tableau, d'entropie $\overline{\mathscr{H}}$, obtenu en fusionnant R_k et R_{k+1}. Alors $r_k+r_{k+1} \geqslant n(\mathscr{H}-\overline{\mathscr{H}})$.*

Démonstration. Posons $\overline{r} = r_k+r_{k+1}$, puis $x = r_k/\overline{r}$ et $y = r_{k+1}/\overline{r}$. Puisque la fonction $t \mapsto -t\log_2(t)$ est concave, on constate alors comme prévu que

$$n(\mathscr{H}-\overline{\mathscr{H}}) = -\overline{r}\,(x\log_2(x)+y\log_2(y)) \leqslant -\overline{r}(x+y)\log_2\left((x+y)/2\right) = \overline{r}.$$

□

Implémentation en pratique.

Au vu de la présentation de ShiversSort adaptatif en page et des implémentations actuelles de TimSort en Python et en Java, trois questions se posent naturellement.

(1) Quelle est la taille maximale de la pile $\mathscr{P}$?

Au vu de l'invariant (1), on sait que, dans une pile $\mathscr{P}$ de taille $h \geqslant 3$, on a $\ell_h \geqslant \ell_3+(h-3) \geqslant h-3$. Puisque $\ell_h \leqslant \log_2(n)$, on en déduit que la pile $\mathscr{P}$ sera de taille $h \leqslant \log_2(n)+3$.

En pratique, puisque les petits *runs* sont déjà fusionnés par une méthode *ad hoc*, les tailles des piles déjà utilisées en Python et en Java sont en fait suffisantes.

(2) On peut manifestement fusionner les cas n° 1 et 2 ; peut-on faire mieux ?

Il se trouve que la réponse est positive. En effet, on peut montrer que, à tout moment de l'exécution de l'algorithme, les prochains *runs* fusionnés seront les *runs* les plus à gauche (disons R_i et R_{i+1}) tels que $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$; ici, par convention, on a posé $\ell_{\rho+1} = +\infty$, de sorte qu'un tel entier i existe toujours. Par conséquent, on peut en fait omettre le cas n° 3, et ce sans changer les fusions qu'exécutera l'algorithme ni l'ordre dans lequel elles seront exécutées.

(3) Comment tester efficacement si $\ell_i \leqslant \ell_j$?

Au vu de la remarque précédente, il nous faudrait en fait tester si $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$. Dans ce cas, il suffit en fait d'exécuter le programme suivant, dont on laisse au lecteur le plaisir de démontrer qu'il nous fournit un résultat correct.

Entrée : Longueurs r_i, r_{i+1} et r_{i+2}
Résultat : `vrai` si $\ell_i \leqslant \max\{\ell_{i+1}, \ell_{i+2}\}$, et `faux` sinon
Remarque: Les opérations **et**, **ou** et **non** sont des opérations bit-à-bit.

$x \leftarrow r_{i+1}$ **ou** r_{i+2}
renvoyer $x \geqslant ((\textbf{non}\ x)\ \textbf{et}\ r_i)$

Toutes ces remarques illustrent la facilité que l'on aurait à remplacer l'utilisation de TimSort par celle de ShiversSort adaptatif. Ainsi, parmi les 908 lignes de l'implémentation actuelle de TimSort en Java (version 13), il suffirait de remplacer les lignes 405 à 412 par le code minimaliste suivant :

```
int n = stackSize - 3;
int x = runLen[n+1] | runLen[n+2];
if (n < 0 || ((~x) & runLen[n]) > x) {
    break;
}
```

Références

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud et Carine Pivoteau. *On the worst-case complexity of Timsort.* 26th Annual European Symposium on Algorithms (ESA), LIPIcs vol. 112, pages 4 :1–13, 2018.

[2] Nicolas Auger, Cyril Nicaud et Carine Pivoteau. *Merge strategies : from merge sort to Timsort.* Rapport technique HAL : hal-01212839, 2015.

[3] Jérémy Barbay et Gonzalo Navarro. *On compressing permutations and adaptive sorting.* Theoretical Computer Science, pages 513 :109–123, 2013.

[4] Sam Buss et Alexander Knop. *Strategies for stable merge sorting.* 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1272–1290. SIAM, 2019.

[5] Stijn De Gouw, Jurriaan Rot, Frank de Boer, Richard Bubel et Reiner Hähnle. *OpenJDK's Java.utils.Collection.sort() is broken : The good, the bad and the worst case.* 27th International Conference on Computer Aided Verification (CAV), pages 273–289. Springer, 2015.

[6] Edsger Dijkstra. *Smoothsort, an alternative for sorting in situ.* Theoretical Foundations of Programming Methodology, pages 3–17. Springer, 1982.

[7] Vladmir Estivill-Castro et Derick Wood. *A survey of adaptive sorting algorithms.* ACM Computing Surveys, vol. 24, issue 4, pages 441–476, 1992.

[8] Herman Goldstine et John von Neumann. *Planning and coding of problems for an electronic computing instrument.* 1947.

[9] Mordecai Golin et Robert Sedgewick. *Queue-mergesort.* Information Processing Letters, vol. 48, issue 5, pages 253–259, 1993.

[10] Tony Hoare. *Algorithm 64 : Quicksort.* Communications of the ACM, vol. 4, issue 7, page 321, 1961.

[11] Vincent Jugé. *Adaptive Shivers sort : an alternative sorting algorithm.* 31th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). SIAM, 2020.

[12] Donald Knuth. *The Art of Computer Programming, Volume 3 (2nd ed.) – Sorting and Searching*. Addison Wesley Longman Publish. Co., Redwood City, CA, USA, 1998.

[13] Heikki Mannila. *Measures of presortedness and optimal sorting algorithms.* IEEE Transactions on Computers, vol. 34, issue 4, pages 318–325, 1985.

[14] Alistair Moffat, Gary Eddy et Ola Petersson. *Splaysort : Fast, versatile, practical.* Software : Practice and Experience, vol. 26, issue 7, pages 781–797, 1996.

[15] J. Ian Munro et Sebastian Wild. *Nearly-optimal mergesorts : Fast, practical sorting methods that optimally adapt to existing runs.* 26th Annual European Symposium on Algorithms (ESA 2018), LIPIcs vol. 112, pages 63 :1–15, 2018.

[16] Tim Peters. *TimSort description.* `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.

[17] Claude Shannon. *A mathematical theory of communication.* Bell system technical journal, vol. 27, issue 3, pages 379–423, 1948.

[18] Olin Shivers. *A simple and efficient natural merge sort.* Rapport de recherche, Georgia Institute of Technology, 2002.

[19] Tadao Takaoka. *Partial solution and entropy.* 34th International Symposium on Mathematical Foundations of Computer Science (MFCS), pages 700–711. Springer Berlin Heidelberg, 2009.

[20] John Williams. *Algorithm 232 : Heapsort.* Communications of the ACM, vol. 7, pages 347–348, 1964.