Réordonnancement par pile

1 Introduction

1.1 Objectifs

Il arrive occasionnellement que l'on ait à réordonner des éléments dans une séquence (typiquement un *flux* de données) de manière « continue », sans que l'on ne puisse disposer, à un instant donné, de l'ensemble des éléments. Par exemple, dans une transmission par internet, les données peuvent être découpées en plusieurs « paquets » qui circulent indépendamment sur le réseau. Ces paquets peuvent parfois suivre des chemins différents et parvenir dans le désordre. Ces paquets sont numérotés afin de pouvoir les remettre dans le bon ordre à l'arrivée. Seulement, s'il s'agit d'un flux de donnée (une vidéo par exemple), on ne peut pas attendre d'avoir la totalité des données pour les trier.

On suppose les éléments dans le flux comparables d'une manière ou d'une autre, et on voudrait, autant que possible, faire en sorte qu'il apparaissent dans le flux sortant *par ordre croissant*. On peut se représenter les choses comme ci-dessous, où des éléments arrivent depuis la droite (ici dans un mauvais ordre, puisque 42 arrive avant 37). On souhaite ainsi ajouter un dispositif visant à réordonner les éléments pour qu'ils poursuivent leur route vers la gauche si possible dans le bon ordre (on voudrait donc que le 37 passe devant le 42 dans le flux sortant).

En fonction des besoins et des informations dont on dispose, il existe plusieurs approches possibles pour réordonner les éléments dans le flux. Nous allons voir une approche utilisant une (ou plusieurs) piles permettant, dans certains cas, d'effectuer un tel réordonnement, et en étudier ses limites.

1.2 Implémentation OCaml

On souhaite, dans ce sujet, modéliser le principe du réordonnancement en OCaml afin de pouvoir l'étudier. On supposera que les éléments dans le flux sont de simples entiers, et donc directement comparables avec l'opérateur `<`.

Pour modéliser les « flux » de données entrant et sortant, on utilisera deux files :

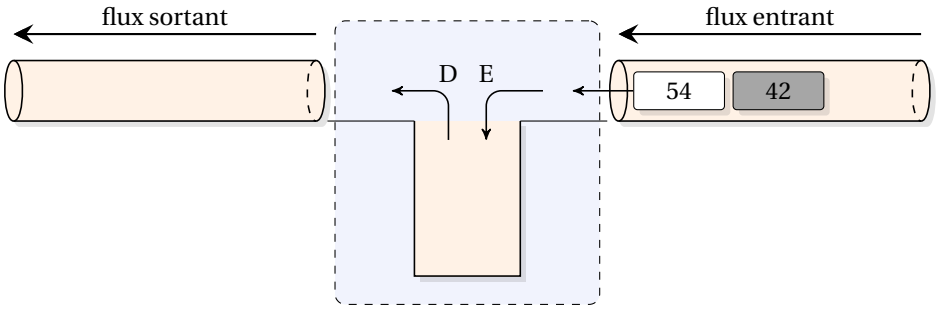
- une file d'entrée, représentée dans le schéma ci-dessus à droite, et dans laquelle on lira les données à trier avec des `Queue.pop` :
- une file de sortie, représentée dans le schéma ci-dessus à gauche, dans laquelle on insérera les données, *si possible correctement triées*, avec des `Queue.push`.

À tout moment, on ne pourra consulter que la prochaine donnée qui arrive (en utilisant `Queue.top` sur la file d'entrée) mais *pas les suivantes*.

2 Séquences triables par pile

2.1 Principe du réordonnancement par pile

Pour tenter de réordonner les données dans le flux, on souhaite interposer entre les deux files, de la sorte :



À tout moment, il est possible d'empiler une donnée extraite de la file d'entrée (on notera cette opération **E** pour « **E**mpiler »), si toutefois la file d'entrée n'est pas vide, ou bien de dépiler une donnée pour l'envoyer dans la file de sortie (on notera cette opération **D** pour « **D**épiler »). **Aucune autre opération n'est permise**.

Par exemple, si la file d'entrée est constituée de 42, puis 37, on peut envisager la suite d'opérations E, E, D, D qui commence par empiler 42, puis 37, puis dépile 37 pour l'envoyer vers la sortie, et enfin dépile 42 pour l'envoyer vers la sortie. Cette séquence d'opérations a effectivement permis de réordonner convenablement les éléments.

1. On suppose avoir n éléments dans la file d'entrée. Combien d'opérations **E** et combien d'opérations **D** sont nécessaires pour faire passer tous les éléments dans la file de sortie ? Toute séquence d'opérations contenant le bon nombre de **E** et de **D** peut-elle être utilisée ?

2. Proposer une séquence d'opérations permettant de réordonner convenablement les éléments si l'on trouve dans la pile d'entrée 37, 17, 29, 54 et 42, qui arrivent dans cet ordre.

3. Trouver une séquence de trois éléments pour laquelle il n'existe aucune séquence d'opérations permettant d'obtenir ces trois éléments dans le bon ordre en sortie. s On souhaite déterminer un algorithme permettant de décider de l'ordre des opérations permettant de réordonner convenablement les éléments lorsque c'est possible.

4. Justifier que si une séquence permet d'avoir les éléments dans le bon ordre dans la

file de sortie, alors à aucun moment dans la pile ne doivent se trouver dans la pile deux éléments tels que celui du dessus est strictement plus grand que celui du dessous.

On se propose donc d'implémenter l'algorithme suivant : s'il est possible d'effectuer l'opération E sans contrevenir à la condition précédente, alors on effectue l'opération **E**. Sinon, on effectue l'opération **D**. Pour l'implémenter en OCaml, on propose l'ébauche de fonction suivante :

```
let sort q_in =
  let q_out = Queue.create () in
  let s = Stack.create () in
  let e () = Stack.push (Queue.pop q_in) s in    (* opération E *)
  let d () = Queue.push (Stack.pop s) q_out in   (* opération D *)
  while ... do
    if ... then e () else d ()
  done;
  q_out
```

Cette fonction prend en argument une file entrante, crée une nouvelle file sortante et une pile, et a pour objectif de vider intégralement la file entrante, remplir la pile sortante, en effectuant soit des opérations **E**, soit des opérations **D**. Elle renvoie la file sortante.

5. Compléter la fonction en déterminant les expressions booléennes adéquates à placer dans le `while` et le `if`.

2.2 Test de la fonction

Pour simplifier les tests dans la suite, on fournit deux fonctions. Une première fonction `queue_of_array` prenant en argument un tableau et remplissant une file avec les éléments du tableau (**sans toucher au contenu du tableau**) de gauche à droite :

```
let queue_of_array t =
  let q = Queue.create () in
  Array.iter (fun x -> Queue.push x q) t;
  q
```

Et une seconde fonction `array_of_queue` prenant en argument une file, **la vidant de ses éléments** puis créant et remplissant un tableau avec ceux-ci, tableau renvoyé par la fonction [1] :

```
let array_of_queue q =
  Array.init (Queue.length q) (fun i -> Queue.pop q)
```

1. On se permet exceptionnellement d'utiliser `Queue.length` ici pour que la fonction ne soit pas trop complexe.

6. Tester la fonction `sort` pour différentes séquences. Grâce aux fonctions fournies, on pourra par exemple écrire :

```
array_of_queue (sort (queue_of_array [|37; 17; 29; 54; 42|]))
```

Le résultat devrait être le tableau `[|17; 29; 37; 42; 54|]`. On vérifiera également que certaines séquences (dont celle trouvée précédemment) ne sont *pas* intégralement triées.

2.3 Analyse de la fonction

7. Quelle est la complexité en temps, dans le pire des cas, de la fonction `sort` lorsqu'elle traite une séquence de n éléments ? La complexité en espace ?

8. Montrer que si une séquence peut être correctement réordonnée avec une séquence bien choisie d'opérations **E** et **D**, alors il existe une unique séquence d'opérations qui convienne.

9. Montrer aussi soigneusement que possible que si une séquence peut être correctement réordonnée avec une séquence bien choisie d'opérations **E** et **D**, alors la fonction précédente y parvient.

3 Permutations triables par pile

3.1 Permutations d'entiers

On dit qu'une séquence $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ de n éléments est *triable par pile* si et seulement si il existe une séquence d'opérations **E** et **D**, telles que définies précédemment, qui permette de les réordonner correctement.

Dans la suite, on s'intéressera plus précisément au cas où $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ est une permutation de l'ensemble des n premiers entiers naturels, $[\![0\,..\,n-1]\!]$ (on peut se convaincre qu'étudier ce cas permet d'étudier le cas de n'importe quelle séquence d'éléments distincts totalement ordonnés).

On souhaite déterminer combien de permutations de $[\![0\,..\,n-1]\!]$ sont triables par pile. Pour ce faire, il nous faut tout d'abord être en mesure de les construire !

On définit un ordre total $\preceq_L$ dit *ordre lexicographique* sur les permutations de $[\![0\,..\,n-1]\!]$ et les tableaux les représentant défini par :

$$\texttt{t1} \prec_L \texttt{t2} \;\equiv\; \exists j \in [\![0\,..\,n-1]\!] \mid \forall i < j,\ \texttt{t1[i]} = \texttt{t2[i]} \text{ et } \texttt{t1[j]} < \texttt{t2[j]}$$

Par exemple, $\langle 2,5,0,3,1,4\rangle \prec_L \langle 2,5,0,4,3,1\rangle \prec_L \langle 3,5,0,4,2,1\rangle$.

10. Quelle est la plus petite permutation possible pour $\preceq_L$ de $[\![0\,..\,n-1]\!]$? La plus grande ?

Pour toute autre permutation que la précédente, on peut vouloir chercher celle qui la suit immédiatement dans l'ordre lexicographique.

11. Déterminer les cinq permutations suivant immédiatement la permutation ci-dessous :

arr	1	2	5	3	0	4

Pour déterminer la permutation suivante, on propose l'algorithme suivant :
- déterminer le plus grand index j tel que `arr.(j)<arr.(j+1)` ;
- déterminer l'indice $k > j$ tel que `arr.(k)>arr.(j)` et `arr.(k)` soit le plus petit possible ;
- échanger le contenu des cases d'index j et k ;
- renverser le contenu des cases d'index $j+1$ à $n-1$ (par renverser, on entend échanger le contenu des cases $j+1$ et $n-1$, des cases $j+2$ et $n-2$ et ainsi de suite).

12. Proposer une fonction `swap` de signature `'a array -> int -> int -> unit` prenant en argument un tableau et deux index de cases valides et échangeant le contenu des deux cases désignées par les deux index.

13. Proposer une fonction `next` de signature `'a array -> bool` appliquant l'algorithme précédent à un tableau `arr` : elle devra renvoyer `false` si la permutation dans le tableau est la plus grande pour l'ordre lexicographique, et `true` si ce n'est pas le cas, auquel cas elle modifiera le contenu du tableau pour qu'il contienne la permutation immédiatement suivante pour l'ordre lexicographique.

3.2 Dénombrement des permutations triables par pile

14. Proposer une fonction `sorted` de signature `'a Queue -> bool` et renvoyant un booléen indiquant si la file contient des éléments ordonnés par ordre croissant. La fonction peut vider (totalement ou partiellement) la file en effectuant le test.

15. Construire, à partir des éléments précédents, une fonction `non_sortable` de signature `int -> int` prenant en argument un entier n et renvoyant le nombre de permutations de $[\![0\,..\,n-1]\!]$ qui ne peuvent pas être réordonnées par `sort`.

16. Utiliser la fonction précédente pour déterminer le nombre de permutations de $[\![0\,..\,n-1]\!]$ ne pouvant pas être réordonnées par `sort` pour n entre 2 et 7 inclus.

17. En déduire, pour ces mêmes valeurs de n, le nombre, noté c_n, de permutations de $[\![0\,..\,n-1]\!]$ pouvant être réordonnées par `sort`. Reconnaissez-vous cette séquence ? Si ce n'est pas le cas, utiliser le site `www.oeis.org` pour la retrouver.

Soient $1 \leqslant p \leqslant n$ deux entiers, $a_0, a_1, \ldots, a_{n-1}$ une séquence de longueur n et $b_0, b_1, \ldots b_{p-1}$ une séquence de longueur p. On dit que la séquence $b_0, b_1, \ldots b_{p-1}$ est un motif de $a_0, a_1, \ldots, a_{n-1}$ s'il existe $i_0 < i_1 < \ldots < i_{p-1}$ tels que $b_0, b_1, \ldots b_{p-1}$ et $a_{i_0}, a_{i_1}, \ldots a_{i_{p-1}}$ soient isomorphes pour la relation d'ordre.

Par exemple, $\langle 0,2,1,3\rangle$ est un motif de $\langle 2,0,5,1,8,4,3,6,7\rangle$ car $\langle 0,2,1,3\rangle$ et $\langle 1,4,3,7\rangle$ sont isomorphes.

18. Montrer qu'une séquence $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ est triable par pile si et seulement si $\langle 1,0,2\rangle$ n'est un motif de $\langle a_0, a_1, \ldots, a_{n-1}\rangle$.

19. En déduire que le nombre c_n de permutations de $[\![0\,..\,n-1]\!]$ vérifie la relation de récurrence

$$c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$$

4 Piles en série

4.1 Mise en série de dispositifs de réordonnancement

On remarquera que la fonction `sort` prend en argument une file et renvoie une file. On peut donc essayer d'appliquer plusieurs fois de suite la fonction `sort` à une file pour essayer de trier une séquence.

20. Proposer une fonction `multisort` de signature `int -> 'a Queue.t -> 'a Queue.t` prenant en argument un entier k positif ou nul et une file (qui pourra éventuellement être vidé par la fonction), appliquant k fois la fonction `sort` et renvoyant une file résultant des k applications successives (si $k = 0$, il est permis de renvoyer la file fournie en argument telle quelle).

On définit le nombre d'inversions dans une séquence $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ comme le nombre de couples $0 \leqslant i < j < n$ tels que $a_i > a_j$.

21. Quel est le nombre minimal de permutations dans une séquence à n éléments ? Le nombre maximal ?

22. On note $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ la séquence renvoyée par `sort` si l'on lui donne en entrée la séquence $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. Que peut-on dire du nombre d'inversions dans $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ par rapport au nombre d'inversions dans $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ si ce dernier n'est pas nulle ? Et s'il est nul ?

23. En déduire qu'il existe un k tel que, quelle que soit la permutation de $[\![0\,..\,n-1]\!]$ fournie à la fonction « `multisort k` », celle-ci renverra une séquence correctement ordonnée. Proposer une borne supérieure pour k.

On souhaite savoir, pour un n donné, quel est le plus petit k qui garantisse que toute séquence de longueur n soit correctement ordonnée par un appel à « multisort k ».

24. Proposer une fonction `non_multisortable` de signature `int -> int -> int` prenant en argument n et k et renvoyant le nombre de permutations de $[\![0\,..\,n-1]\!]$ qui ne peuvent pas être correctement ordonnées par `multisort` pour le k choisi.

25. Utiliser la fonction précédente pour $n = 7$ et k entre 0 et 10.

26. Proposer une fonction `opt_k` de signature `int -> int` qui prend en argument un entier strictement positif n, et, à partir de la fonction précédente, détermine k_n, le plus

petit k qui garantisse que « `multisort k` » sache réordonner correctement n'importe quelle permutation de $[\![0\,..\,n-1]\!]$.

27. Utiliser la fonction précédente pour n entre 5 et 9. Que peut-on conjecturer pour k_n ?

28. En s'aidant des fonctions écrites, trouver plusieurs permutations de $[\![0\,..\,6]\!]$ qui ne peuvent pas être réordonnées avec `multisort` pour $k = k_7 - 1$. Que constate-t-on ?

29. En s'aidant des exemples précédents, prouver la conjecture.

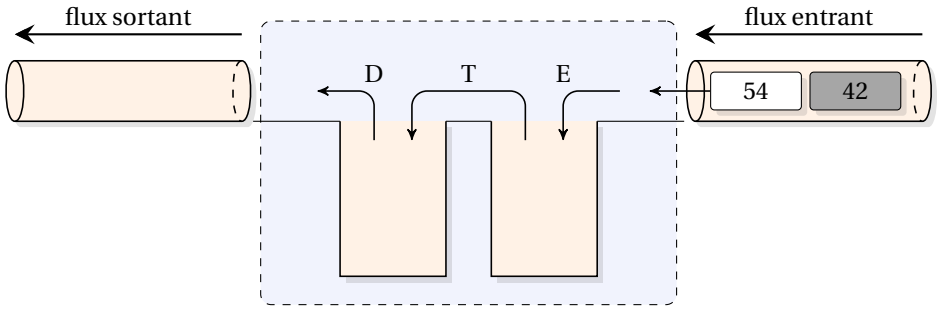
30. On a donc la possibilité de construire une méthode de tri pour un tableau à n éléments (en utilisant `multisort` avec $k = k_n$). Quelle est sa complexité ?

5 Doubles piles

5.1 Objectif

31. Déterminer les deux permutations de $[\![0\,..\,3]\!]$ qui ne sont pas ordonnables par « `multisort 2` » (soit deux passages consécutifs dans un système de réordonnancement par pile). N'hésitez pas à le faire expérimentalement !

On souhaite savoir si on peut faire mieux en utilisant plus intelligemment deux piles. On considère donc le système suivant :



On dispose à présent de trois opérations : il est possible d'empiler une donnée extraite de la file d'entrée dans la pile de droite (**E** pour « **E**mpiler »), de dépiler une donnée de la pile de gauche pour l'envoyer dans la file de sortie (**D** pour « **D**épiler ») et enfin de dépiler un élément de la pile de droite pour l'empiler sur la pile de gauche (**T** pour « **T**ransférer »). **Aucune autre opération n'est permise**.

32. Proposer, pour chacune des deux séquences trouvées dans la question précédente, une séquence d'opérations **E**, **D** et **T** permettant d'obtenir le bon ordre en sortie, illustrant qu'un usage plus élaboré des deux piles permet de réordonner davantage de séquences.

33. Montrer que si la séquence est triable, alors les éléments dans une des deux piles doivent être à tout moment ordonnés, mais que ce n'est pas le cas dans l'autre pile.

5.2 Approche gloutonne pour deux piles

Il est difficile de trouver une stratégie simple permettant de réordonner efficacement les données d'un flux avec deux piles sans hypothèses supplémentaires. On va se restreindre au cas particulier où l'on sait que les éléments dans le flux sont exactement les entiers de 0 à $n-1$ (donc une permutation de $[\![0\,..\,n-1]\!]$), sans pour autant connaître n à l'avance.

On propose pour le réordonnancement à deux piles l'algorithme suivant :

- si k éléments ont déjà été envoyés vers la file sortante et que l'élément k se trouve au sommet de la pile de gauche, alors on effectue **D** ;
- sinon, si c'est possible sans violer la condition exposée à la question précédente, on effectue **T** ;
- sinon, si la file d'entrée n'est pas vie, on effectue **E** ;
- sinon, enfin, on applique **D**.

On propose une ébauche OCaml de cette fonction :

```
let sort2 q_in =
  let q_out = Queue.create () in
  let sl = Stack.create () in       (* Pile de gauche *)
  let sr = Stack.create () in       (* Pile de droite *)
  let count = ref 0 in              (* Nombre d'éléments sortis *)
  let e () = Stack.push (Queue.pop q_in) sr in
  let d () = (Queue.push (Stack.pop sl) q_out; incr count) in
  let t () = Stack.push (Stack.pop sr) sl in
  while ... do
    if ... then d ()
    else if ... then t ()
    else if ... then e ()
    else d ()
  done;
  q_out
```

34. Compléter la fonction précédente avec les quatre expressions booléennes adéquates.

35. Proposer une fonction `non_2sortable` de signature `int -> int` prenant en argument un entier n et renvoyant le nombre de permutations de $[\![0\,..\,n-1]\!]$ qui ne peuvent pas être ordonnées par `sort2`.

36. Déterminer expérimentalement le nombre de permutations de $[\![0\,..\,n-1]\!]$ qui ne peuvent pas être réordonnées avec `sort2` pour $n \in [\![3\,..\,9]\!]$. Le comparer avec le nombre de permutations ne pouvant pas être réordonnées avec « `multisort 2` ».

37. Déterminer les permutations de $[\![0\,..\,4]\!]$ qui ne peuvent pas être réordonnées par `sort2`. Est-il possible de trouver une séquence d'opérations **E**, **D** et **T** permettant d'obtenir le bon ordre en sortie pour ces permutations problématiques ?