Chapitre 5

Tris

Ce chapitre traite des algorithmes de tri. La première section présente trois algorithmes de tri interne pour lesquels aucune hypothèse particulière n'est faite sur les clés : le tri rapide, le tri fusion et le tri par tas. La seconde section concerne le tri externe. On y présente un algorithme de construction des monotonies et deux algorithmes de répartition : le tri équilibré et le tri polyphasé.

Introduction

Dans ce chapitre, nous présentons trois algorithmes de tri parmi les plus efficaces lorsqu'aucune hypothèse particulière n'est faite sur la nature ou la structure des clés des éléments à trier. Le *tri rapide* place définitivement un élément appelé pivot, construit une liste «gauche» des éléments dont la clé est inférieure ou égale à celle du pivot, construit une liste «droite» des éléments dont la clé est strictement supérieure à celle du pivot et réalise enfin un appel récursif sur chacune des deux listes. Le *tri fusion* est également un tri dichotomique. Il réalise d'abord un appel récursif sur chacune des deux listes obtenues par scission de la liste initiale en deux listes de taille égale (à une unité près) et interclasse ensuite les deux listes triées. Le *tri par tas* place d'abord les éléments à trier dans un tableau organisé en file de priorité (tas), puis supprime un à un les éléments de priorité minimum du tas tout en les replaçant simultanément dans le tableau dans l'ordre inverse. Ces trois algorithmes de tri ont une complexité moyenne $O(n \log n)$. Dans le pire des cas, le tri rapide a pour complexité $O(n^2)$, alors que la complexité des deux autres tris est $O(n \log n)$. D'autres algorithmes moins performants comme par exemple les tris simples (tri bulle, tri par insertion, tri par sélection,...) ne sont pas présentés ici mais ont, pour la plupart d'entre eux, servi d'illustration à d'autres sujets traités dans le livre. Certains algorithmes, par exemple le *tri par champs*, sont plus performants mais exigent certaines hypothèses spécifiques sur les clés des éléments à trier. Nous proposons en exercice l'étude de certains algorithmes de ce type.

5.1 Tri interne

La donnée d'un problème de tri est constituée d'une permutation $L = (e_1, \ldots, e_n)$, d'un ensemble E de n éléments et d'une application $c : E \mapsto F$ de E dans un ensemble F totalement ordonné qui associe à chaque élément e une *clé* $c(e)$. Nous appellerons élément maximal de E un élément dont la clé est le plus grand élément de $c(E)$. Inversement, un élément minimal de E est un élément dont la clé est le plus petit élément de $c(E)$. Une *liste triée* de E est une permutation de E dont les éléments sont rangés dans un ordre compatible avec la relation d'ordre sur F. Trier la liste L, c'est déterminer une liste triée des éléments de L.

Exemple. Si $E = \{A, B, C, D\}$, $F = \{0, 1\}$, $c(A) = c(B) = 1$ et $c(C) = c(D) = 0$, la liste (C, D, B, A) est une liste triée.

Nous mesurerons la complexité d'un tri par le nombre de comparaisons de clés qu'il réalise et la taille d'un problème de tri sera définie par le nombre n d'éléments à trier. Nous supposerons également que la liste initiale L est implémentée par un tableau linéaire $T[i..j]$ où $n = j - i + 1$. Ces hypothèses sont réalistes dans le cas d'un *tri interne* où les éléments à trier sont et restent disponibles en mémoire centrale. La section 5.2 traitera le cas du tri externe pour lequel cette hypothèse n'est pas satisfaite.

5.1.1 Le tri rapide

Le tri rapide d'une liste L sur E est un algorithme comprenant deux phases. La première phase transforme la liste L en une liste $L' = G \cdot (\pi) \cdot D$ sur E où π est un élément de L appelé *pivot*, G est la *liste gauche* contenant g éléments ($g \leq n-1$), D est la *liste droite* contenant d éléments ($d \leq n-1$). De plus, la clé d'un élément quelconque de G est inférieure ou égale à la clé d'un élément quelconque de D.

Notons que chacune des listes G ou D peut être vide et qu'elles le sont toutes les deux si $n = 1$. Plus précisément, si le pivot est un élément maximal, alors la liste D est vide, si le pivot est un élément minimal, alors la liste G peut être vide ou non. Il est important de remarquer qu'après la première phase, le pivot est bien placé et que les tailles de la liste gauche et de la liste droite sont strictement plus petites que n.

La seconde phase consiste simplement en deux appels récursifs de l'algorithme, l'un pour la liste G, l'autre pour la liste D.

Un appel de l'algorithme de tri rapide est *terminal* si $n \leq 1$. Notons qu'un appel terminal n'effectue aucune comparaison de clés. Nous pouvons donc décrire l'algorithme de tri rapide par la procédure TRI RAPIDE ci-dessous. La clé de l'élément $T(k)$ est notée $c(k)$ et les listes associées aux appels récursifs correspondent à des sous-tableaux de $T[i..j]$. La première phase correspond à la fonction PIVOTER qui retourne l'indice du pivot.

```
procédure TRI RAPIDE(T[i..j]);
  si i < j alors
    k :=PIVOTER(T[i..j]);
    TRI RAPIDE(T[i..(k − 1)]);
    TRI RAPIDE(T[(k + 1)..j])
  finsi.
```

L'algorithme de pivotage

L'efficacité du tri rapide dépend directement de l'algorithme de pivotage. Ce dernier doit d'une part choisir le pivot et d'autre part réorganiser le tableau.

Nous conviendrons de choisir comme pivot le *premier élément du tableau*, c'est-à-dire l'élément $T(i)$ dont la clé $c(i)$ est notée conventionnellement γ. Ce choix, qui n'est pas le seul possible, n'est pas plus mauvais qu'un autre en l'absence d'hypothèse supplémentaire sur les données et présente le double avantage d'être simple à implémenter et de conduire à une analyse en moyenne rigoureuse.

Nous appelons *couple inversé* du tableau par rapport au pivot un couple d'indices (s, t) tel que :

$$i < s < t \leq j, \quad c(s) > \gamma, \quad c(t) \leq \gamma.$$

Le couple inversé (s, t) *précède* le couple inversé (s', t') si $s' \geq s$ et $t' \leq t$. Cette relation de précédence confère à l'ensemble des couples inversés une relation d'ordre total. Si cet ensemble n'est pas vide, il existe donc un premier couple inversé. La réorganisation du tableau $T[i..j]$ est un algorithme itératif où chaque itération concerne des sous-tableaux emboîtés de T et *toujours la même valeur* de γ.

La première itération concerne le tableau $T[i+1..j]$. Elle consiste à rechercher le premier couple inversé (s'il existe) par un appel à la fonction PREMIER-COUPLE-INVERSÉ qui retourne un couple (s, t). Si $s < t$, alors (s, t) est effectivement le premier couple inversé par rapport à γ, les éléments de rang s et t sont échangés dans le tableau T par la procédure ECHANGER et l'itération suivante concernera le sous-tableau $T[s'..t']$ où $s' = s + 1$ et $t' = t - 1$. L'algorithme se termine si $s' > t'$.

```
fonction PREMIER-COUPLE-INVERSÉ(T, k, l, γ) :couple;
  s := k; t := l;
  tantque s ≤ l etalors c(s) ≤ γ faire s := s + 1 fintantque;
  tantque t ≥ k etalors c(t) > γ faire t := t − 1 fintantque
  PREMIER-COUPLE-INVERSÉ :=(s, t).
```

La fonction PIVOTER ci-dessous implémente l'algorithme de pivotage.

```
fonction PIVOTER(T, i, j) :indice;
  γ :=c(i); s := i + 1; t := j;
  tantque s ≤ t faire
    (s, t) :=PREMIER-COUPLE-INVERSÉ(T, s, t, γ);
    si s < t alors
      ECHANGER(T, s, t); s := s + 1; t := t − 1
    finsi
  fintantque;
  PIVOTER :=t;
  ÉCHANGER(T, i, t).
```

La figure 1.1 montre une exécution de l'algorithme de pivotage. La première ligne contient la liste initiale.

6	(14)	3	1	10	5	1	6	4	5	2	11	9	[6]
6	6	3	1	(10)	5	1	6	4	5	[2]	11	9	14
6	6	3	1	2	5	1	6	4	[5]	(10)	11	9	14
5	6	3	1	2	5	1	6	4	6	10	11	9	14

(○) pointeur gauche
[□] pointeur droit
pivot placé

Figure 1.1: *Un pivotage.*

Le lemme suivant établit un encadrement du nombre de comparaisons de clés réalisées lors d'un pivotage.

Lemme 1.1. *L'algorithme de pivotage réalise au moins $n-1$ et au plus $n+1$ comparaisons de clés.*

Preuve. Si tous les éléments de $T[i+1..j]$ ont une clé inférieure ou égale à γ, la valeur finale de s est $j+1$ et t n'a pas été modifié. Il en résulte $n-1$ comparaisons à partir de s et une comparaison à partir de t, soit en tout n comparaisons. Il y a de même n comparaisons si tous les éléments de $T[i+1..j]$ ont une clé strictement supérieure à γ. Si le tableau $T[i+1..j]$ contient un élément de clé inférieure ou égale à γ et un élément de clé strictement supérieure à γ, deux cas sont possibles. Si le *dernier* échange d'un couple inversé concerne deux éléments contigus dans T, il y a eu exactement $n-1$ comparaisons avant cet échange et il n'y en aura pas après. Sinon, lors de la terminaison du pivotage, les clés des éléments indicés par les dernières valeurs de s et t ont été comparées *deux fois* à γ, il y a donc eu $n+1$ comparaisons (voir figure 1.2). ∎

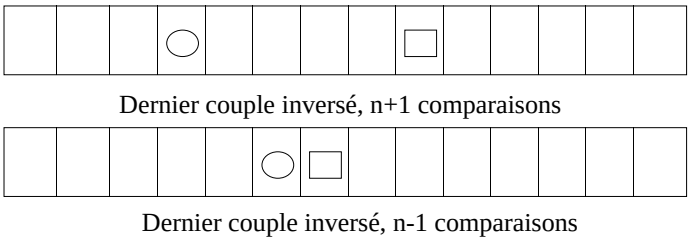

Figure 1.2: *Dernier échange d'un couple inversé.*

Complexité dans le pire des cas

Examinons l'arborescence $\mathcal{A}$ des appels récursifs d'une exécution de l'algorithme de tri rapide. L'arborescence associée à la liste initiale de la figure 1.1 est donnée sur la figure 1.3. Les indices i et j d'un appel sont inscrits à l'intérieur du sommet correspondant. Les sommets des appels terminaux sont cerclés de noir. Cette

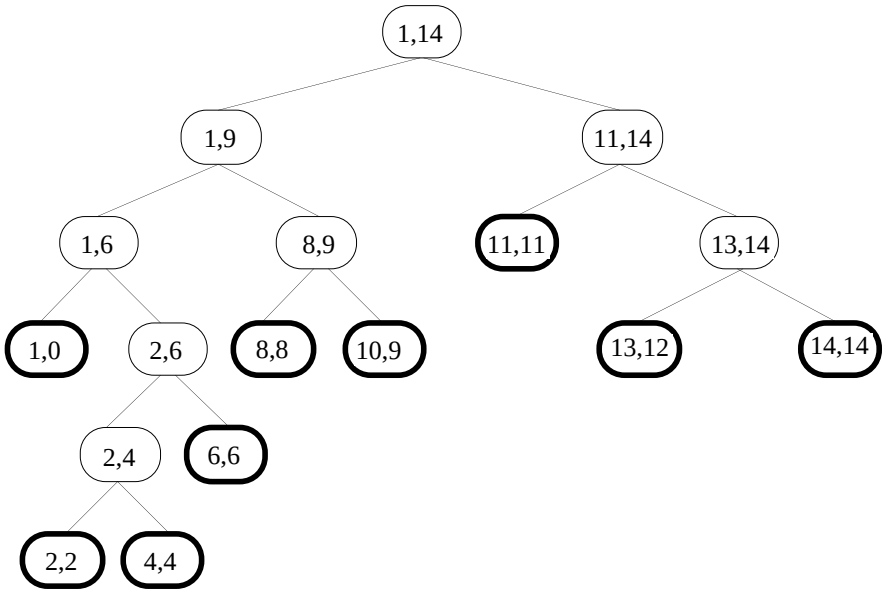

Figure 1.3: *Appels récursifs du tri rapide.*

arborescence est un arbre binaire complet dont les feuilles correspondent à un tableau contenant au plus un élément. Si les deux fils y et z d'un noeud x correspondent à deux tableaux de p et q éléments, le noeud x correspond lui à un tableau de $p+q+1$ éléments. D'après le lemme 1.1, le nombre maximum de comparaisons faites au noeud x , égal à $p+q+2$, est la somme du nombre maximum de comparaisons $(p+1)$ faites au noeud y et du nombre maximum de comparaisons $(q+1)$ faites au noeud z. Le nombre total de comparaisons faites pour tous les noeuds *d'un même niveau* de l'arbre est donc majoré par $n+1$ et le nombre total de comparaisons est lui-même majoré par $(n+1)h(\mathcal{A})$ où $h(\mathcal{A})$ est la hauteur en nombre de sommets de l'arborescence $\mathcal{A}$.

Si la liste initiale contient n éléments, la hauteur maximale est atteinte (voir exercices) lorsque chaque niveau (le niveau 0 de la racine excepté) est constitué de deux noeuds frères, un noeud terminal correspondant à une liste vide et un noeud correspondant à une liste de p éléments si son père est lui-même associé à une liste de $p+1$ éléments (voir figure 1.4). La complexité dans le pire des cas de l'algorithme de tri rapide est donc $O(n^2)$.

Cette complexité est effectivement atteinte lorsque la liste initiale L est une liste triée. En effet l'arborescence induite est alors le «peigne» représenté sur la figure 1.4. Dans ce cas, le nombre de comparaisons (voir lemme 1.1), inscrit à l'intérieur du sommet sur la figure, est *exactement* n pour le niveau 0, $n-1$ pour le niveau 1, $n-i$ pour le niveau i, 2 pour l'avant-dernier niveau et 0 pour le dernier niveau, soit en tout $\sum_{j=2}^{n} j{=}n(n+1)/2-1$ comparaisons.

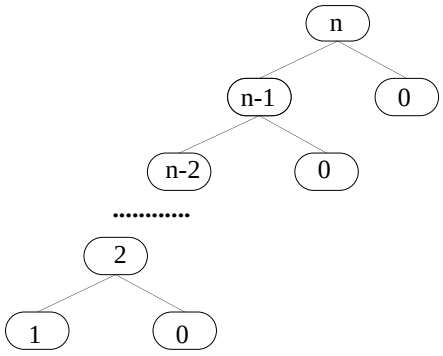

Figure 1.4: *Un «peigne».*

Complexité en moyenne du tri rapide

Nous faisons ici l'hypothèse que les clés des n éléments de la liste initiale L sont *distinctes* et que les $n!$ permutations qui représentent les rangs de ces clés dans L sont *équiprobables*. Nous noterons $\pi(L)$ la permutation des rangs des clés de la liste L.

Exemple. Si $L=(A,B,C,D,E,F)$, $c(A)=3$, $c(B)=1$, $c(C)=5$, $c(D)=10$, $c(E)=7$ et $c(F)=15$, on a $\pi(L)=(2,1,3,5,4,6)$. Le lemme suivant montre que l'algorithme de pivotage conserve les hypothèses probabilistes de la liste initiale pour les deux sous-listes gauche et droite.

Lemme 1.2. *Soit r le rang du pivot. Après l'algorithme de pivotage, les permutations de $\{1,\ldots,r-1\}$ associées aux listes gauches G sont équiprobables. Il en est de même pour les permutations de $\{r+1,\ldots,n\}$ associées aux listes droites D.*

Preuve. La preuve est immédiate si $r=1$ ou si $r=n$. Nous supposons donc $1<r<n$ et nous choisissons une permutation possible en sortie de l'algorithme

de pivotage, notée $\pi = \pi_1 \cdot (r) \cdot \pi_2$, où π_1 est une permutation de $\{1, \ldots, r-1\}$ et π_2 est une permutation de $\{r+1, \ldots, n\}$.

Supposons que l'algorithme de pivotage ait réalisé à partir d'une liste L, les q échanges de couples inversés $(s_1, t_1), (s_2, t_2), \cdots, (s_q, t_q)$ et que π soit la permutation des rangs de la liste obtenue après pivotage. On a alors :

$$1 \leq s_1 < s_2 < \cdots < s_q \leq r-1 \qquad r+1 \leq t_q < t_{q-1} < \cdots < t_1 \leq n. \tag{1.1}$$

Si maintenant on opère sur la permutation π la transposition τ_0 qui consiste à échanger $\pi(1)$ et $\pi(r)$ puis les q transpositions τ_k, $k = 1, .., q$, où τ_k consiste à échanger $\pi(s_k)$ et $\pi(t_k)$, on retrouve la permutation des rangs de la liste L.

Réciproquement, si nous choisissons q entiers naturels s_k et q entiers naturels t_k satisfaisant (1.1), et si nous opérons sur π la transposition τ_0 suivie des q transpositions τ_k, nous obtenons une permutation π' qui après pivotage donne la permutation π. De plus, et c'est là le point crucial, deux choix distincts des entiers s_k et t_k satisfaisant (1.1) correspondent à deux permutations π' distinctes.

Pour r, π et q fixés, le nombre de choix possibles des entiers s_k et t_k satisfaisant 1.1 est $\binom{r-1}{q}\binom{n-r}{q}$. Comme les valeurs possibles pour q sont celles de l'ensemble $\{1, \ldots, \min\{r-1, n-r\}\}$, le nombre total de permutations qui, après pivotage, donne la permutation π est

$$\sum_{q=1}^{\min\{r-1,n-r\}} \binom{r-1}{q}\binom{n-r}{q}.$$

Ce nombre ne dépend que du rang r du pivot et non de la permutation π elle-même. Il en résulte que, pour r fixé, les permutations π après pivotage sont équiprobables. Il en est bien sûr de même des permutations π_1 et π_2. ∎

Le lemme précédent permet à l'analyse en moyenne de pouvoir profiter de la structure récursive de l'algorithme de tri rapide. En effet l'hypothèse d'équiprobabilité des permutations $\pi(L)$ est conservée pour les deux appels récursifs. Nous notons M_n le nombre *moyen* de comparaisons de clés réalisées pour une liste initiale de n éléments. En conditionnant par la valeur r du rang du pivot (les n valeurs de r sont équiprobables), les lemmes 1.2 et 1.1 nous permettent d'écrire pour $n \geq 2$:

$$M_n \leq \sum_{r=1}^{n} 1/n \, [n + 1 + M(r-1) + M(n-r)].$$

Les conditions initiales sont $M(0) = M(1) = 0$ puisqu'aucune comparaison n'est exécutée pour un appel terminal. En développant l'inégalité précédente et en remarquant que

$$\sum_{r=1}^{n} M(r-1) = \sum_{r=1}^{n} M(n-r) = \sum_{r=1}^{n-1} M(r)$$

il vient :

$$M_n \leq n + 1 + 2/n \sum_{r=1}^{n-1} M(r).$$

Pour obtenir une majoration de $M(n)$, nous résolvons l'équation de récurrence :

$$S_n = n + 1 + 2/n \sum_{r=1}^{n-1} S(r)$$

avec $S(0) = S(1) = 0$. La solution de cette équation de récurrence complète, développée dans la section 2.4 du chapitre 2, est donnée par

$$S_n = 2(n + 1)(H_{n+1} - 4/3),$$

où H_n est le $n^{ième}$ nombre harmonique. Il en résulte que $M_n = O(n \log n)$.

L'analyse de l'algorithme de pivotage a également montré que le nombre de comparaisons d'un pivotage est au moins $n-1$. On montre alors à partir de l'inégalité

$$M_n \geq \sum_{r=1}^{n} 1/n[n - 1 + M(r - 1) + M(n - r)]$$

et en suivant la même démarche que pour la majoration de M_n, que

$$M(n) \geq 2(n + 1)H_n - 4n.$$

Il en résulte que $M(n) = \theta(n \log n)$.

5.1.2 Le tri fusion

Le principe de l'algorithme de tri fusion est de scinder la liste initiale L de n éléments en deux sous-listes G et D, ayant respectivement $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ éléments, telles que $L = G \cdot D$. Un appel récursif est réalisé pour chacune des deux listes G et D. L'*interclassement* des deux listes triées fournit ensuite la liste triée résultat.

La complexité du tri fusion dépend donc de l'algorithme d'interclassement. Etant données deux listes triées L_1 et L_2 ayant respectivement n et m éléments, il est possible de réaliser leur interclassement dans un tableau résultat R en temps $O(n+m)$ (voir exercices). Dans le tri-fusion on peut profiter du fait que les deux listes se trouvent côte-à-côte dans le tableau initial pour construire un algorithme d'interclassement plus raffiné que celui du cas général mais qui a la même complexité en temps. La procédure INTERCLASSER ci-dessous recopie dans la partie gauche du tableau de travail la liste L_1 et dans la partie droite la liste L_2 inversée. Ce placement astucieux des deux listes permet alors de replacer dans le tableau initial T les $n + m$ éléments du plus petit au plus grand en répétant $n + m$ fois le même traitement. La liste L_1 occupe initialement dans T les positions de g à m et la liste L_2 les positions de $m + 1$ à d.

```
procédure INTERCLASSER(T, g, m, d);
  pour i de g à m faire R(i) := T(i);
  pour j de m + 1 à d faire R(j) := T(d + m + 1 − j);
  i := g; j := d;
  pour k de g à d faire
    si R(i) < R(j)
      alors T(k) := R(i); i := i + 1
      sinon T(k) := R(j); j := j − 1
    finsi
  finpour.
```

L'algorithme du tri-fusion est alors implémenté par la procédure TRI-FUSION ci-dessous. Cette procédure devra utiliser un tableau de travail *global* R pour réaliser les interclassements.

```
procédure TRI-FUSION(T, i, j);
  si i < j alors
    n := j − i + 1; g := i; m := i + ⌊n/2⌋ − 1; d := j;
    TRI-FUSION(T, g, m);
    TRI-FUSION(T, m + 1, d);
    INTERCLASSER(T, g, m, d)
  finsi.
```

La figure 1.5 montre l'arborescence des appels récursifs du tri fusion pour la liste de la figure 1.1. Dans chaque sommet sont inscrits en partie haute les bornes du sous-tableau et en partie basse le sous-tableau résultat de cet appel. Les sommets des appels terminaux sont arrondis.

Complexité du tri fusion dans le pire des cas

Soit n le nombre d'éléments du tableau initial. Le premier appel récursif de la procédure TRI-FUSION porte sur un tableau de $\lfloor n/2 \rfloor$ éléments et le second sur un tableau de $\lceil n/2 \rceil$ éléments. Comme le nombre maximum de comparaisons de la procédure INTERCLASSER pour deux sous-tableaux contenant en tout n éléments est $n-1$, le nombre maximum de comparaisons pour un tableau initial de taille n, noté $f(n)$, satisfait pour $n \geq 2$ l'inégalité :

$$f(n) \leq n - 1 + f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil).$$

La condition initiale est $f(1) = 0$. Nous obtenons une majoration de f en résolvant l'équation de récurrence (de partitions)

$$g(n) = n - 1 + g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil),$$

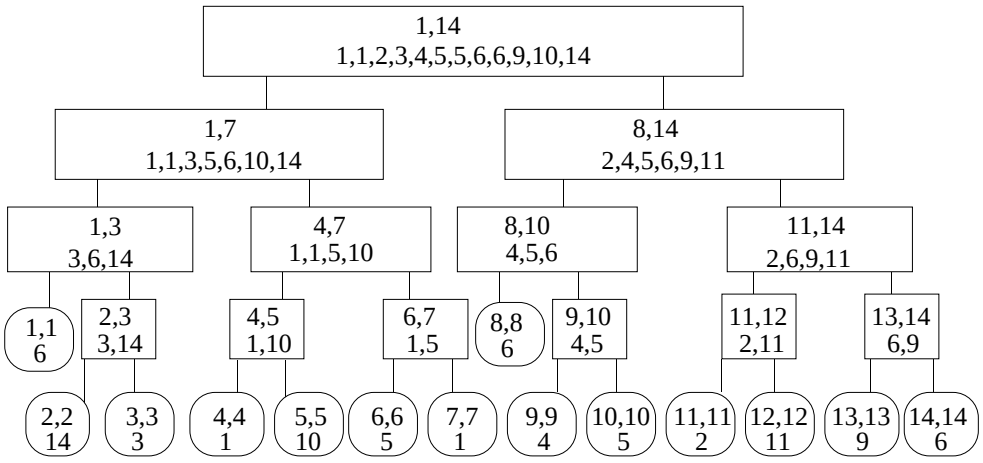

Figure 1.5: *Un tri fusion.*

avec $g(1) = 0$ comme condition initiale. Cette équation étudiée dans la section 2.3 du chapitre 2 (voir aussi l'exercice 2.4 du chapitre 2) a pour solution :

$$g(n) = \begin{cases} 0 & \text{si } n = 1 \\ n\lceil \log n \rceil + 1 - 2^{\lceil \log n \rceil} & \text{si } n \geq 2 \end{cases}$$

Il en résulte que la complexité $f(n)$ du tri fusion dans le pire des cas est $O(n \log n)$.

Le nombre minimum de comparaisons d'un interclassement de deux listes de taille m et n est $\min\{m, n\}$. Dans le cas de la procédure TRI-FUSION, ce nombre est $\lfloor n/2 \rfloor$. Le nombre minimum de comparaisons, noté $h(n)$ d'un tri fusion d'une liste de n éléments vérifie donc l'inégalité :

$$h(n) \geq \lfloor n/2 \rfloor + h(\lfloor n/2 \rfloor) + h(\lceil n/2 \rceil).$$

On obtient une minoration de h en résolvant l'équation de récurrence

$$g(n) = \begin{cases} 0 & \text{si } n = 1 \\ \lfloor n/2 \rfloor + g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil) & \text{si } n \geq 2. \end{cases}$$

Les résultats sur les récurrences de partition permettent alors de montrer que $g(n) = \theta(n \log n)$. La complexité du tri fusion est donc $\theta(n \log n)$.

5.1.3 Le tri par tas

Rappelons qu'un tas est un arbre tournoi parfait dont les sommets contiennent les éléments d'un ensemble E muni d'une clé $c : E \mapsto F$ où F est totalement ordonné. Dans la section 3.4, des algorithmes ont été donnés pour l'insertion et pour la suppression d'un élément de plus petite clé. Ces algorithmes utilisent une implémentation du tas par un couple (T, p) où p est le nombre d'éléments contenus dans le tas et T est un tableau à p positions. Le tri par tas consiste simplement à

insérer un à un les n éléments de la liste initiale puis à réaliser n suppressions d'un élément de plus petite clé. Afin de pouvoir utiliser directement les algorithmes INSÉRERTAS et EXTRAIREMIN donnés dans la section 3.4, nous supposerons que la liste initiale est implémentée par le tableau $T[1..n]$. La procédure TRI-PAR-TAS ci-dessous utilise uniquement le tableau T pour toutes les opérations de mise à jour.

```
procédure TRI-PAR-TAS(T, n);
   p := 0; {création d'un tas vide}
   pour k de 1 à n faire INSÉRER(T(k), T);
   pour k de 1 à n faire EXTRAIREMIN(T).
```

Avant la $k^{ième}$ insertion, le tas est constitué des $k-1$ premiers éléments de T et les éléments non encore insérés occupent les $n-k$ positions suivantes. Après la $k^{ième}$ suppression, le tas occupe les $n-k$ premières positions du tableau et les éléments occupant les k dernières positions sont rangés par clé décroissante au sens large. En effet, l'échange des contenus de la racine et de la dernière feuille du tas, réalisé au début de EXTRAIREMIN(T), permet de « récupérer » l'élément supprimé lors de la $k^{ième}$ suppression dans la position $n-k+1$ du tableau.

Comme la complexité en temps des algorithmes INSÉRER et EXTRAIREMIN est $O(\log n)$ où n est le nombre d'éléments du tas, la complexité du tri par tas est $O(n \log n)$. La figure 1.6 montre l'exécution de l'algorithme de tri par tas. Une barre oblique à l'intérieur du tableau indique la séparation entre le tas en partie gauche et les éléments non contenus dans le tas en partie droite.

5.2 Tri externe

Les problèmes de tri étudiés jusqu'ici supposent que tous les objets à trier sont placés en mémoire centrale. Cependant en pratique, cette hypothèse ne peut être satisfaite en raison de la taille et du nombre des objets à trier. Ces objets sont alors placés sur des mémoires secondaires (disques, bandes magnétiques, etc.) et l'algorithme de tri exécutera des opérations de lecture et d'écriture sur le support utilisé. Comme la durée d'une opération d'entrée-sortie sur une mémoire externe est beaucoup plus grande qu'un temps d'accès à la mémoire centrale, le temps opératoire d'un algorithme de *tri externe* est essentiellement dû au nombre d'opérations d'entrée-sortie qu'il exécute. Ce nombre sera la mesure de complexité que nous utiliserons pour ce type d'algorithme.

Le mode d'accès (séquentiel, direct, etc.) est bien sûr un paramètre important pour la conception d'un algorithme de tri externe. Nous nous limiterons au cas de supports identiques à *accès séquentiel*.

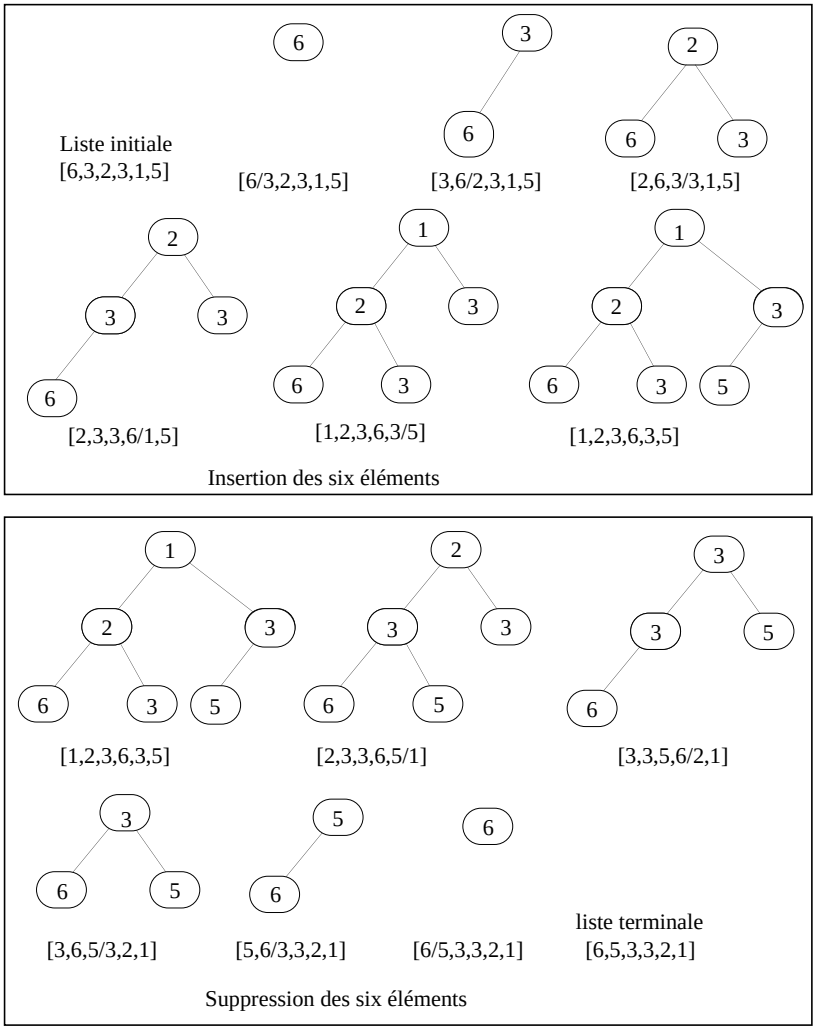

Figure 1.6: *Un tri par tas.*

La structure générale d'un algorithme de tri externe comprend la résolution de trois problèmes. Le premier est la *construction de monotonies*, c'est-à-dire de sous-listes disjointes triées. Le second est la *répartition dynamique* des monotonies sur les supports. Le troisième est l'*interclassement* des monotonies. Les solutions retenues pour ces trois problèmes doivent concourir à l'obtention sur un support de la liste triée de tous les objets en minimisant le nombre d'opérations d'entrée-sortie pour y parvenir. Il faut en particulier éviter des situations dites «de blocage» où toutes les monotonies se trouvent placées sur un même support alors que le tri n'est pas terminé.

Dans ce chapitre nous étudierons plus spécialement le problème de la répartition dynamique des monotonies et l'une de ses solutions, appelée *tri polyphasé*, qui, en utilisant les nombres de Fibonacci, conduit à un algorithme performant. Nous noterons s le nombre de supports, n le nombre d'objets à trier et m la place disponible en mémoire centrale, mesurée en nombre d'objets. On suppose que les

n objets sont initialement placés sur un même support noté S_0.

5.2.1 Construction des monotonies

Pour limiter le nombre d'entrées-sorties, il convient de construire un petit nombre de monotonies de grande taille. Deux techniques conduisant à un nombre comparable d'entrées-sorties répondent à cet objectif. L'une construit des monotonies de même taille m en utilisant un algorithme de tri interne; l'autre construit des monotonies en général plus longues par sélection et remplacement.

Monotonies de même taille

Chaque itération consiste à lire en mémoire centrale les m objets suivants du support S_0 (ou le nombre restant d'objets s'il est inférieur à m), à trier les objets contenus en mémoire centrale par un algorithme de tri interne et à écrire la liste obtenue sur un support qui dépend de la stratégie de répartition. L'algorithme réalise ainsi d'une part $2n$ opérations d'entrée-sortie (n lectures et n écritures) et d'autre part $\lceil n/m \rceil$ tris internes.

Sélection et remplacement

L'algorithme est initialisé par la lecture en mémoire centrale des m premiers objets de S_0. On note E l'ensemble des objets contenus en mémoire centrale et f l'objet suivant sur S_0. L'ensemble E contient un sous-ensemble M initialement vide d'objets dits *marqués* car ils ne peuvent appartenir à la monotonie en cours de construction. Le calcul de la monotonie suivante commence lorsque M est vide et se termine lorsque M est égal à E. Une itération consiste à calculer le plus petit élément e de $E-M$, à placer e à la suite de la monotonie courante en construction et à lire en mémoire centrale l'objet f de S_0 dans la place occupée par e. Si f est plus petit que e, il est ajouté à l'ensemble M car il ne pourra pas faire partie de la monotonie courante. Le pire des cas correspond aux objets rangés initialement dans l'ordre inverse de l'ordre requis. Après la lecture en mémoire centrale des m premiers objets, tout nouvel objet lu en mémoire est immédiatement marqué. Chaque monotonie est alors de taille m. Dans le cas général, la taille de chaque monotonie (sauf peut-être la dernière) est plus grande que m. Si les objets non marqués en mémoire centrale sont organisés en tas, la complexité des opérations élémentaires hors entrées-sorties est $O(n \log m)$ et le nombre d'opérations d' entrée-sortie est $2n$ puisque chaque élément est lu et écrit une seule fois. La figure 2.1 décrit les dix premières étapes d'un exemple de construction de monotonies par sélection et remplacement pour une mémoire de 5 cellules (m=5). La liste initiale sur support externe est S_0=$(4, 3, 6, 7, 10, 5, 1, 2, 8, 3, 5, 9, 4)$. L'élément de la ligne i et de la colonne j est la clé de l'objet contenu dans la cellule i à l'étape j. Si cette clé est suivie d'une étoile, la cellule i appartient à M. Si cette clé est

suivie d'une pointe de flèche, l'objet correspondant est le suivant de la monotonie en cours. Comme après l'étape 8 toutes les cellules sont marquées, la première monotonie est $(3, 4, 5, 6, 7, 8, 10)$. La seconde commence par $(1, 2, \ldots)$.

	1	2	3	4	5	6	7	8	9	10
1	4	4^	1*	1*	1*	1*	1*	1*	1^	4
2	3^	5	5^	2*	2*	2*	2*	2*	2	2^
3	6	6	6	6^	8	8^	5*	5*	5	5
4	7	7	7	7	7^	3*	3*	3*	3	3
5	10	10	10	10	10	10	10^	9*	9	9

changement de monotonie

Figure 2.1: *Sélection et remplacement.*

5.2.2 Répartition des monotonies

Une fois construite, chaque monotonie doit être placée sur l'un des s supports. La stratégie de répartition des monotonies sur les supports est ici essentielle pour l'efficacité des opérations ultérieures d'interclassement. Nous examinons deux stratégies de répartition. Le *tri équilibré* a le mérite d'être simple à implémenter, mais utilise mal les supports, réalise $2n$ opérations d'entrée-sortie par phase et surtout est contraint d'exécuter des recopies de monotonies. Le *tri polyphasé*, en fondant sa stratégie de répartition sur les nombres de Fibonacci, réalise un nombre limité d'opérations d'entrée-sortie à chaque phase, n'est jamais contraint à de simples recopies, mais doit par contre introduire des *monotonies fantômes* si le nombre total de monotonies initiales n'est pas l'un des éléments d'une suite spécifique déduite de la suite de Fibonacci d'ordre $s - 1$.

Le tri équilibré

Le tri équilibré consiste à définir deux sous-ensembles $\mathcal{L}$ (supports de lecture) et $\mathcal{E}$ (supports d'écriture) de q supports chacun. Lors de leur création, les monotonies sont réparties *uniformément* sur les supports de $\mathcal{L}$. Une *phase* de l'algorithme est décrite par la procédure PHASE-TRI-ÉQUILIBRÉ ci-dessous. L'itération fondamentale de cette procédure, appelée *fusion élémentaire* et implémentée par FUSION(σ, k), lit la première monotonie de chaque support de σ, interclasse ces monotonies en mémoire centrale et écrit la monotonie résultat sur le support d'écriture k.

```
procédure PHASE-TRI-ÉQUILIBRÉ;
  k := 0  (mod q);
  tantque toutes les monotonies de L n'ont pas été lues faire
    soit σ le sous-ensemble des supports non vides de L;
    FUSION(σ, k);
    k := k + 1  (mod q)
  fintantque;
  L := ensemble des supports de E;
  E := ensemble des supports de L.
```

Une phase du tri équilibré fait passer le nombre de monotonies de M à $\lceil M/q \rceil$ et les nouvelles monotonies restent uniformément réparties sur les supports. Si M_0 est le nombre initial de monotonies, le nombre maximal de phases est en $O(\log M_0)$. Chaque phase réalise $2n$ entrées-sorties car chaque objet est lu et écrit une fois. Il en résulte que le nombre d'opérations d'entrée-sortie du tri équilibré est en $O(n \log M_0)$. La figure 2.2 décrit un tri équilibré. Les M_i sont les monotonies initiales et f est l'opérateur de fusion élémentaire. Les sous-ensembles $\{S_1, S_2, S_3\}$ et $\{S_4, S_5, S_6\}$ sont alternativement utilisés en lecture et en écriture. Lors de la phase 1, la monotonie M_3 a simplement été recopiée.

	phase 0	phase 1	phase 2
S1	M1,M2,M3		f(f(M1,M4,M6),f(M2,M5,M7),M1)
S2	M4,M5		
S3	M6,M7		
S4		f(M1,M4,M6)	
S5		f(M2,M5,M7)	
S6		M3=f(M3)	

Figure 2.2: *Le tri équilibré.*

Comme nous l'avons souligné, le principal inconvénient du tri équilibré est de réaliser une lecture et une écriture de chaque objet lors d'une même phase. De plus, si en fin de phase tous les supports de lecture sont vides sauf un ne contenant qu'une seule monotonie, celle-ci sera simplement recopiée et le nombre de monotonies ne diminuera pas lors de cette fusion élémentaire.

Le tri polyphasé

Par rapport au tri équilibré, le tri polyphasé ne réalise au cours d'une phase qu'un nombre limité d'opérations d'entrée-sortie, n'exécute pas de simples recopies et

utilise beaucoup mieux l'ensemble des supports en se servant des nombres de Fibonacci pour répartir initialement les monotonies.

L'itération fondamentale du tri polyphasé, également appelée *phase*, consiste à choisir un support dit d'écriture, noté e et à réaliser k fusions élémentaires sur le support e. L'*état du système* est défini par la suite ordonnée par valeurs décroissantes $N = (N_1, \ldots, N_s)$ des nombres de monotonies sur les S supports, chaque rang de cette suite correspondant à un numéro logique de support. On passe des numéros logiques aux numéros physiques par une permutation et l'on appelle *total de* N le nombre $\tau(N) = \sum_{i\in\{1,\ldots,s\}} N_i$. Le lemme suivant détermine pour un état terminal N fixé, quel est l'état initial de plus grand total permettant d'atteindre N *en une seule phase*.

Lemme 2.1. *L'état* $(N_1 + N_2, N_1 + N_3, \cdots, N_1 + N_s, 0)$ *est un état de total maximal permettant d'atteindre l'état* N *en une seule phase.*

Preuve. Notons $\psi(N) = (N_1 + N_2, N_1 + N_3, \cdots, N_1 + N_s, 0)$. En prenant comme suppport d'écriture e celui qui ne contient aucune monotonie et en réalisant N_1 fusions élémentaires de l'ensemble des autres supports sur e, on construit une phase qui fait passer de l'état $\psi(N)$ à l'état $(N_2, N_3, \cdots, N_s, N_1)$ qui, à une permutation près, est l'état N. Remarquons de plus que $\tau(\psi(N)) = \tau(N)+(s-2)N_1$. Supposons maintenant que M soit un état initial permettant de passer en une seule phase de l'état M à l'état N. Considérons les numéros logiques des supports dans M. Soit e le numéro du support d'écriture choisi et $a_i, i \in \{1, \ldots, s\} - \{e\}$ le nombre d'occurrences du support numéro i dans les k fusions élémentaires de la phase. Le nombre de monotonies du support numéro i devient $M'_i = M_i - a_i, i \in \{1, \ldots, s\} - \{e\}$ et celui du support numéro e devient $M'_e = M_e + k$. Comme nous avons $k \leq M'_e$ et pour tout i dans $\{1, \ldots, s\} - \{e\}$, $a_i \leq k$, il vient :

$$\tau(M) = \tau(M') + (\sum_{i\in\{1,\ldots,s\}-\{e\}} a_i) - k \leq \tau(M') + (s-2)k \leq \tau(N) + (s-2)N_1$$

car $\tau(M') = \tau(N)$ et $k \leq N_1$. ∎

Soit $E^1 = (1, 0, \cdots, 0)$ l'état final associé à la liste des objets totalement triée sur un seul support. Il résulte du lemme précédent que $\psi^{(n)}(E^1)$ est une répartition initiale sur les supports qui permet d'obtenir l'état E^1 en n phases. Si m est le nombre initial de monotonies, le tri polyphasé utilise comme répartition initiale le vecteur $\psi^{(k)}(E^1)$ où k est le plus petit entier tel que $\tau(\psi^{(k)}(E^1)) \geq m$. Il faudra donc ajouter et répartir sur les supports $m - \tau(\psi^{(k)}(E^1))$ *monotonies fantômes*.

Répartition initiale des monotonies

Nous examinons dans cette section la nature et les propriétés essentielles de la suite $\psi^{(n)}(E^1)$, $n \in \mathbb{N}$ et la répartition initiale des monotonies fantômes. Notons que n est l'indice générique de la suite et ne représente pas dans cette section le

nombre d'éléments à trier. Nous notons $F_s^n = (f^n{}_1, \ldots, f^n{}_s)$ le vecteur $\psi^{(n)}(E^1)$ et $T_s(n)$ son total. Il résulte directement de la définition de ψ que F_s^{n+1} est obtenu à partir de F_s^n par la relation de récurrence suivante :

$$f_i^{n+1} = \begin{cases} f_1^n + f_{i+1}^n & \text{si } i \leq s-1 \\ 0 & \text{si } i = s \end{cases}$$

avec comme conditions initiales :

$$f_i^0 = \begin{cases} 1 & \text{si } i = 1 \\ 0 & \text{sinon} \end{cases}$$

Le tableau ci-dessous montre les valeurs de F_s^n pour $s = 6$ et $0 \leq n \leq 8$.

n	f_1^n	f_2^n	f_3^n	f_4^n	f_5^n	f_6^n	$T(n)$
0	1	0	0	0	0	0	1
1	1	1	1	1	1	0	5
2	2	2	2	2	1	0	9
3	4	4	4	3	2	0	17
4	8	8	7	6	4	0	33
5	16	15	14	12	8	0	65
6	31	30	28	24	16	0	129
7	61	59	55	47	31	0	253
8	120	116	108	92	61	0	497

Dans le cas particulier de trois supports, l'équation de récurrence précédente s'écrit pour $n \geq 2$:

$$f_1^{n+1} = f_1^n + f_2^n \qquad f_2^{n+1} = f_1^n.$$

Sa solution est immédiate car nous avons pour $n \geq 2$:

$$f_1^{n+1} = f_1^n + f_1^{n-1} \qquad f_2^{n+1} = f_1^{n-1} + f_2^{n-1} = f_2^n + f_2^{n-1}.$$

Compte tenu des conditions initiales, à savoir $f_1^0 = f_1^1 = 1$, $f_2^0 = 0$ et $f_2^1 = 1$, nous obtenons :

$$f_1^n = \varphi(n+1) \qquad f_2^n = \varphi(n) \qquad T_3(n) = \varphi(n+2)$$

où $\varphi(n)$ est l'élément de rang n de la suite de Fibonacci d'ordre deux définie par :

$$\varphi(n) = \begin{cases} \varphi(n-1) + \varphi(n-2) & \text{si } n \geq 2 \\ 1 & \text{si } n = 1 \\ 0 & \text{si } n = 0 \end{cases}$$

Dans le cas général, la suite de Fibonnaccci d'ordre p, notée $\Phi^p(n)$ est définie par :

$$\Phi^p(n) = \begin{cases} \sum_{k=1}^{p} \Phi^p(n-k) & \text{si } n \geq p \\ 1 & \text{si } n = p-1 \\ 0 & \text{si } n \in \{0, \ldots, p-2\} \end{cases}$$

On montre alors que dans le cas de s supports :

$$f_k^n = \Phi^{s-1}(n+(s-1)-2)+\Phi^{s-1}(n+(s-1)-3)+\cdots+\Phi^{s-1}(n+(s-1)-(s+1-k)$$

et que le nombre total de monotonies est donné par :

$$T_s(n) = (s-1)\Phi^{s-1}(n+(s-3)) + (s-2)\Phi^{s-1}(n+(s-4)) + \cdots + \Phi^{s-1}(n-1)$$

Supposons que le nombre initial m de monotonies soit compris strictement entre $T_s(n)$ et $T_s(n+1)$. Il faut donc créer $T_s(n+1)-m$ monotonies fantômes que l'on a avantage à répartir aussi uniformément que possible sur les supports. L'algorithme FIBONACCI(s,m) ci-dessous réalise cet objectif en construisant la répartition F_s^{n+1} et en donnant la priorité aux monotonies fantômes.

```
procédure FIBONACCI(s, m);
  {On suppose connu l'entier n tel que T_s(n) < m < T_s(n+1)}
  r := T_s(n+1) − T_s(n); F := T_s(n+1) − m;
  pour S de 1 à s − 1 faire d(S) := f_S^{n+1} − f_S^n;
  S := 0;
  pour j de 1 à r faire
    S := S + 1  (mod s − 1);
    si d(S) > 0 alors
      d(S) := d(S) − 1;
      si j < F
        alors écrire une monotonie fantôme sur le support S + 1
        sinon écrire la monotonie suivante sur le support S + 1;
      finsi
    finsi
  finpour;
  pour S de 1 à s − 1 faire
    écrire les f_S^n monotonies suivantes sur le support S
  finpour.
```

La figure 2.3 montre la répartition réalisée par l'algorithme précédent pour 19 monotonies initiales et 6 supports. Pour cet exemple, on a : $T_6(3)$=17, $T_6(4)$=33 et $F_6(4)$=$(8,8,7,6,4,0)$.

Analyse du tri polyphasé

La fonction génératrice $\hat{\Phi}^p$ de la suite de Fibonacci d'ordre p est donnée par :

$$\hat{\Phi}^p(z) = \sum_{n\in\mathbb{N}} \Phi^p(n) z^n = \frac{z^{p-1}}{1-z-z^2-\cdots-z^p}.$$

S1	f	f	f	m	m	m	m	m
S2	f	f	f	m	m	m	m	m
S3	f	f	f	m	m	m	m	
S4	f	f	f	m	m	m		
S5	f	f	m	m				

f: monotonie fantôme
m: monotonie réelle

Figure 2.3: *Répartition des monotonies fantômes.*

Il résulte alors des formules du paragraphe précédent donnant les f_p^n et $T_s(n)$ que les fonctions génératrices $\hat{f}_1$ et $\hat{T}_s$ des suites f_1^n et $T_s(n)$ sont respectivement :

$$\hat{f}_1(z) = \sum_{n\in\mathbb{N}} f_1(n)z^n = \frac{1}{1-z-z^2-\cdots-z^p}$$

et

$$\hat{T}_s(z) = \sum_{n\in\mathbb{N}} T_s(n)z^n = \frac{(s-1)z+(s-2)z^2+\cdots+z^{s-1}}{1-z-z^2-\cdots-z^p}.$$

Comme les zéros du polynôme $1-z-z^2-\cdots-z^p$ sont tous de module strictement plus petit que l'unité, on a $T_s(n) = O(a^n)$ où a est un rationnel strictement plus grand que l'unité. Si donc $m = T_s(n)$ est le nombre initial de monotonies, le *nombre de phases* du tri polyphasé est en $O(\log m)$.

Pour évaluer le nombre d'opérations d'entrée-sortie qui, rappelons-le, est notre mesure de complexité, nous supposons que toutes les monotonies initiales ont la même taille t et nous introduisons la notion de *transfert*. Un transfert correspond au «passage» de t objets d'un support sur un autre. Nous allons compter le nombre de transferts réalisés par un tri polyphasé à partir de la distribution initiale F_s^n. Notons que ce calcul prend en compte la taille des monotonies construites au cours de l'exécution du tri. Le tableau ci-dessous, construit pour un exemple à huit phases et six supports, montre l'évolution de la taille des monotonies sur chaque support, du nombre de monotonies sur chaque support, et du nombre de transferts t par phase.

n	S_1	S_2	S_3	S_4	S_5	S_6	t
1	$[1]^{61}$	$[1]^{59}$	$[1]^{55}$	$[1]^{47}$	$[1]^{31}$	$[0]$	253
2	$[5]^{31}$	$[1]^{30}$	$[1]^{28}$	$[1]^{24}$	$[1]^{16}$	$[0]$	5×31
3	$[9]^{16}$	$[5]^{15}$	$[1]^{14}$	$[1]^{12}$	$[1]^{8}$	$[0]$	9×16
4	$[17]^{8}$	$[9]^{8}$	$[5]^{7}$	$[1]^{6}$	$[1]^{4}$	$[0]$	17×8
5	$[33]^{4}$	$[17]^{4}$	$[9]^{4}$	$[5]^{3}$	$[1]^{2}$	$[0]$	33×4
6	$[65]^{2}$	$[33]^{2}$	$[17]^{2}$	$[9]^{2}$	$[5]^{1}$	$[0]$	65×2
7	$[129]^{1}$	$[65]^{1}$	$[33]^{1}$	$[17]^{1}$	$[9]^{1}$	$[0]$	129×1
8	$[253]^{1}$	$[0]$	$[0]$	$[0]$	$[0]$	$[0]$	253×1

Un élément $[a]^b$ du tableau où a et b sont deux entiers doit être interprété comme b monotonies de taille a. Les nombres de Fibonacci apparaissent partout dans ce tableau. On montre que d'une manière générale pour n phases et s supports, la première ligne du tableau central est :

$$([T_s(0)]^{f_1^n}, [T_s(0)]^{f_2^n}, \cdots, [T_s(0)]^{f_{s-1}^n})$$

et que la ligne associée à la phase p (hormis le nombre de transferts) est :

$$([T_s(p-1)]^{f_1^{n-p+1}}, [T_s(p-2)]^{f_2^{n-p+1}}, \cdots, [T_s(p-s+1)]^{f_{s-1}^{n-p+1}})$$

en adoptant la convention que $T_s(k) = 1$ si $k \leq 0$. Il résulte alors de la structure de la ligne générale du tableau central que le nombre de transferts associé à la phase p est donné par :

$$f_{s-1}^{n-p+1}[T_s(p-1) + T_s(p-2) + \cdots + T_s(p-s+1)]$$

ce qui s'écrit encore : $f_1^{n-p} \times T_s(p)$. Le nombre total de transferts, noté $\theta_s(n)$, est donc égal à $T_s(n) + \sum_{p=1}^{n} f_1^{n-p} \times T_s(p)$. Il en résulte que $\theta_s(n)$ est égal à la somme de $T_s(n)$ et du coefficient de z^n dans $\hat{f}_1(z)\hat{T}_s(z)$.

Lorsque n tend vers $+\infty$, le rapport $\theta_s(n)/T_s(n)$ est équivalent à une fonction linéaire du type $A_s \ln(T_s(n)) + B_s$ où A_s et B_s ne dépendent que du nombre s de supports. Ce comportement a également été observé expérimentalement pour la répartition issue de la procédure FIBONACCI(s, m) lorsque le nombre m de monotonies initiales (de même taille) tend vers $+\infty$ mais n'est pas nécessairement égal à l'une des valeurs de la suite $T_s(n)$.

Notes

Dans ce chapitre, nous nous sommes limités à la présentation de trois algorithmes classiques de tri interne et un algorithme de tri externe. L'ouvrage le plus complet du domaine est :
D. Knuth, *The Art of Computer Programming*, tome **3** : Sorting and Searching, Addison Wesley, 1973.

Exercices

5.1. Le *tri bulle* d'un tableau $t[i..j]$ de n éléments consiste, pour k variant de i à $j-1$, à faire «remonter» le plus petit élément du sous-tableau $t[k..j]$ en position k par une suite d'échanges. La procédure TRI-BULLE ci-dessous implémente cet algorithme :

```
procédure TRI-BULLE(t, i, j);
  pour k de i à j − 1 faire
    pour p de j − 1 à k pas −1 faire
      si c(p + 1) < c(p) alors ÉCHANGER(t, p + 1, p)
      {c(k) est la clé de l'élément t(k)}
    finpour
  finpour.
```

a) Déterminer le nombre de comparaisons exécutées par le tri bulle sur un tableau à n éléments.
b) Déterminer le nombre maximum et le nombre minimum d'échanges exécutés par le tri bulle sur un tableau à n éléments.
On suppose maintenant que les clés sont distinctes deux à deux et l'on note $\pi(t)$ la permutation de $\{1, \ldots, n\}$ associée au rang des éléments du tableau $t[i..j]$.
c) Démontrer que le nombre d'échanges réalisés par le tri bulle est égal au nombre d'inversions de π.
On note $I_{n,k}$ le nombre de permutations de $\{1, \ldots, n\}$ ayant k inversions. Exprimer $I_{n,k}$ en fonction des $I_{n-1,p}$ où $p \in \{\max\{0, k-n+1\}, \ldots, k\}$.
En déduire que la fonction génératrice $I_n(z) = \sum_{k\in\mathbb{N}} I_{n,k} z^k$ vérifie :

$$I_n(z) = (1 + z + z^2 + \ldots + z^{n-1}) I_{n-1}(z)$$

En supposant que les permutations $\pi(t)$ sont équiprobables, montrer que le nombre moyen d'échanges du tri bulle est $\frac{n(n-1)}{4}$.

5.2. Le *tri par sélection* d'un tableau $t[i..j]$ à n éléments consiste à déterminer l'élément minimum du tableau, à échanger cet élément avec le premier élément du tableau et à réaliser récursivement ces opérations sur le tableau $t[i+1..j]$. La procédure TRI-SÉLECTION ci-dessous implémente cet algorithme :

```
procédure TRI-SÉLECTION(t, i, j);
  si j > i alors
    soit c(k) = min{c(p) | p ∈ {i, ..., j}};
    ÉCHANGER(t, k, i);
    TRI-SÉLECTION(t, i + 1, j)
  finsi.
```

a) Déterminer, en fonction de n, le nombre d'échanges réalisés par la procédure TRI-SÉLECTION.
b) Déterminer, en fonction de n, le nombre minimum et le nombre maximum de comparaisons réalisées par la procédure TRI-SÉLECTION.

c) Déterminer le nombre moyen de comparaisons réalisées par la procédure TRI-SÉLECTION si les clés sont distinctes deux à deux et les permutations des rangs équiprobables.

5.3. Le *tri par insertion* d'un tableau $t[i..j]$ à n éléments consiste à réaliser un appel récursif pour le tableau $t[i..j-1]$ et à insérer le dernier élément du tableau à sa place dans le tableau en décalant d'une position vers la droite les éléments qui le suivent. La procédure TRI-INSERTION ci-dessous implémente cet algorithme :

```
procédure TRI-INSERTION(t, i, j);
  si j > i alors
    TRI-INSERTION(t, i, j − 1);
    k := j;
    tantque k > i etalors c(k) < c(k − 1) faire
      ÉCHANGER(t, k − 1, k)
    fintantque
    {c(k) est la clé de l'élément t(k)}
  finsi.
```

a) Déterminer la complexité du tri par insertion en nombre de comparaisons.
On suppose que les clés sont distinctes deux à deux et que les permutations des rangs sont équiprobables.
b) Démontrer que pour l'appel récursif, les permutations des rangs du tableau $t[i..j-1]$ sont équiprobables.
c) En déduire le nombre moyen de comparaisons du tri par insertion d'un tableau à n éléments.

5.4. Soit E un ensemble de n éléments dont les clés sont des entiers codés avec K chiffres de l'ensemble $\{0,\ldots,9\}$. Le *tri par champs* consiste à construire K listes $\{L_1,\ldots,L_K\}$ où la liste L_k est triée pour les k chiffres de poids faible. La liste L_K est donc une liste triée. On note $p^{(k)}$, $k \in \{1,\ldots,K\}$, le $k^{ième}$ chiffre de poids faible d'un entier p à K chiffres. La procédure TRI-CHAMPS ci-dessous implémente cet algorithme à partir de la liste L des n éléments à trier :

```
procédure TRI-CHAMPS(L);
  pour k de 1 à K faire
    pour i de 1 à n faire
      e :=DÉFILER(L);
      p := c(e)^(k);
      {c(e) est la clé de l'élément e}
      ENFILER(e, L_p)
    finpour;
    pour q de 0 à 9 faire L := L · L_q; L_q := () finpour
  finpour.
```

Les opérations DÉFILER et ENFILER correspondent à la gestion d'une file.
a) Montrer que la liste L obtenue en retour de la procédure TRI-CHAMPS est triée.
b) Exprimer en fonction de n et K la complexité (au sens classique) du tri par champs.
c) En déduire une condition pour que ce tri soit linéaire par rapport à n.

5.5. Soient L_1 et L_2 deux listes triées contenant respectivement n et m entiers et rangées dans deux tableaux T_1 et T_2. Ecrire un algorithme de complexité en temps $O(n+m)$ pour interclasser de ces deux listes dans un tableau T.

5.6. On utilise une variante de l'algorithme de tri rapide pour déterminer le $k^{ième}$ élément d'un ensemble de n éléments munis d'une clé. On suppose que les n clés sont distinctes deux à deux. On considère dans un premier temps la fonction SÉLECT ci-dessous :

```
fonction SÉLECT(t, i, j, k) : élément;
{on suppose que 1 ≤ k ≤ n}
  p :=PIVOTER(t, i, j);
  si k ≤ p
    alors SÉLECT :=SÉLECT(t, i, p, k)
    sinon SÉLECT :=SÉLECT(t, p + 1, j, k − p)
  finsi.
```

où la fonction PIVOTER retourne un entier $p \in \{i+1, \ldots, j-1\}$ et réorganise en temps $O(n)$ le tableau $t[i..j]$ en deux sous-tableaux $t[i..p]$ et $t[p+1..j]$ tels qu'un élément quelconque de $t[i..p]$ possède une clé inférieure ou égale à celle d'un élément quelconque de $t[p+1..j]$.
a) Démontrer que la procédure SÉLECT détermine le $k^{ième}$ élément du tableau.
b) Montrer que, sans hypothèse particulière sur l'algorithme de pivotage, la fonction SÉLECT peut réaliser $O(n^2)$ comparaisons.
c) Montrer que si l'algorithme de pivotage garantit que la taille du sous-tableau de l'appel récursif ne dépasse pas αn où $\alpha < 1$, la complexité en nombre de comparaisons de la fonction SÉLECT est $O(n)$.
On considère l'algorithme de choix du pivot suivant :

1. découper le tableau en $\lfloor n/5 \rfloor$ blocs $\{B_1, \ldots, B_{\lfloor n/5 \rfloor}\}$ de cinq éléments; les éléments restants (au plus 4) ne seront pas considérés dans la suite de l'algorithme;
2. déterminer les éléments médians m_k des B_k, $k \in \{1, \ldots, \lfloor n/5 \rfloor\}$;
3. utiliser la fonction SÉLECT pour déterminer l'élément d'ordre $\left\lfloor \frac{n+5}{10} \right\rfloor$ de la liste $(m_1, \ldots, m_{\lfloor n/5 \rfloor})$; (si $\lfloor n/5 \rfloor$ est pair, l'élément sélectionné est l'élément médian de la liste $(m_1, \ldots, m_{\lfloor n/5 \rfloor})$).

d) Montrer que le pivot choisi est strictement supérieur à au moins $3\left\lfloor \frac{n-5}{10} \right\rfloor$ éléments de t et est inférieur ou égal à au moins $3\left\lfloor \frac{n-5}{10} \right\rfloor$ éléments de t. En déduire

que pour $n \geq 75$, le sous-tableau de l'appel récursif est de taille au plus égale à $3n/4$.

On considère alors la fonction SÉLECT suivante :

```
fonction SÉLECT(t, i, j, k) : élément;
  si (j − i) ≤ 74 alors
    TRIER(t, i, j);
    extraire l'élément d'ordre k de t;
    exit
  sinon
    pour q de 1 à ⌊n/5⌋ faire
      m(q) :=MÉDIAN(t, 5(q − 1) + i, 5(q − 1) + i + 4)
      {m(q) est l'indice dans t de l'élément médian}
      {de {t(5(q − 1) + i), ..., t(5(q − 1) + i + 4)}}
    finpour;
    r :=SÉLECT(m, 1, ⌊n/5⌋ , ⌊(n+5)/10⌋);
    {r est l'indice dans t de l'élément}
    { d'ordre ⌊(n+5)/10⌋ de (t_m(1), ..., t_m(⌊n/5⌋))}
    p :=PARTITION(t, i, j, r);
    si p ≥ k
      alors SÉLECT :=SÉLECT(t, i, p, k)
      sinon SÉLECT :=SÉLECT(t, p + 1, j, k − p)
    finsi
  finsi.
```

La procédure TRIER est un algorithme de tri quelconque. La fonction MÉDIAN fournit l'élément de rang 3 d'une liste de 5 éléments. La procédure PARTITION réorganise le tableau t en prenant $t(r)$ comme pivot ; après son exécution, un élément quelconque de $t[i..p]$ a une clé inférieure ou égale à $t(r)$, un élément quelconque de $t[p+1..j]$ a une clé strictement supérieure à $t(r)$.

e) Démontrer la validité de la fonction SÉLECT.

f) Montrer que la complexité de la fonction SÉLECT est $O(n)$.