Solutions choisies du chapitre 2 : Premiers pas

Solution de l'exercice 2.2-2

```
Tri-SÉLECTION(A)
n = A.longueur
pour j = 1 à n − 1
    minimum = j
    pour i = j + 1 à n
        si A[i] < A[minimum]
            minimum = i
    permuter A[j] et A[minimum]
```

L'algorithme préserve l'invariant de boucle suivant : au début de chaque itération de la boucle **pour** externe, le sous-tableau $A[1 .. j-1]$ contient les $j-1$ plus petits éléments du tableau $A[1 .. n]$ et ce sous-tableau est trié. Après les $n-1$ premiers éléments, le sous-tableau $A[1 .. n-1]$ contient les $n-1$ plus petits éléments, triés, et l'élément $A[n]$ est donc forcément l'élément maximum.

Le temps d'exécution de l'algorithme est $\Theta(n^2)$ dans tous les cas.

Solution de l'exercice 2.2-4

Modifiez l'algorithme de façon qu'il teste si l'entrée respecte une certaine condition de cas spécial et, si oui, sorte une réponse pré calculée. Le temps d'exécution dans le cas le plus favorable n'est généralement pas une bonne mesure pour un algorithme.

Solution de l'exercice 2.3-5

La procédure RECHERCHE-DICHOTOMIQUE prend en entrée un tableau trié *A*, une valeur *v* et une plage [*bas* .. *haut*] du tableau, où nous recherchons la valeur *v*. La procédure compare *v* à l'élément de tableau occupant le milieu de la plage et décide d'éliminer la moitié de la plage de toute considération ultérieure. Nous donnons des versions itérative et récursive, chacune retournant soit un indice *i* tel que $A[i] = v$, soit NIL si aucun élément de [*bas* .. *haut*] ne contient la valeur *v*. L'appel initial à l'une ou l'autre version doit avoir les paramètres *A*, *v*, 1, *n*.

```
RECHERCHE-DICHOTOMIQUE-ITÉRATIVE(A, v, bas, haut)
tant que bas ≤ haut
        moy = ⌊(bas + haut)/2⌋
        si v == A[moy]
            retourner moy
        sinonsi v > A[moy]
            bas = moy + 1
        sinon haut = moy – 1
retourner NIL
```

```
RECHERCHE-DICHOTOMIQUE-RÉCURSIVE(A, v, bas, haut)
si bas > haut
        retourner NIL
    moy = ⌊(bas + haut)/2⌋
    si v == A[moy]
        retourner moy
    sinonsi v > A[moy]
        retourner RECHERCHE-DICHOTOMIQUE-RÉCURSIVE(A, v, moy + 1, haut)
    sinon retourner RECHERCHE-DICHOTOMIQUE-RÉCURSIVE(A, v, bas, moy – 1)
```

Les deux procédures se terminent sur un échec quand la plage est vide (c'est-à-dire quand *bas* > *haut*) et sur un succès quand la valeur *v* a été trouvée. En fonction de la comparaison entre *v* et l'élément médian de la plage de recherche, la recherche continue sur la plage diminuée de moitié. La récurrence pour ces procédures est donc $T(n) = T(n/2) + \Theta(1)$, dont la solution est $T(n) = \Theta(\lg n)$.

Solution du problème 2-4

a. Les inversions sont (1, 5), (2, 5), (3, 4), (3, 5), (4, 5). (Retenez que les inversions sont spécifiées par des indices, et non par les valeurs du tableau.)

b. Le tableau contenant des éléments de $\{1, 2, \ldots, n\}$ qui a le plus d'inversions est $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. Pour tout $1 \le i < j \le n$, il y a une inversion (i, j). Le nombre de ces inversions est $\binom{n}{2} = n(n-1)/2$.

c. Supposez que le tableau *A* commence par une inversion (k, j). Alors $k < j$ et $A[k] > A[j]$. Au moment où la boucle **pour** externe des lignes 1–8 fait *clé* = $A[j]$, la valeur de départ dans $A[k]$ est encore quelque part à gauche de $A[j]$. En clair, elle est dans $A[i]$, avec $1 \le i < j$, et l'inversion est donc devenue (i, j). Une certaine itération de la boucle **tant que** des lignes 5–7 décale $A[i]$ d'une position vers la droite. La ligne 8 finira par déposer *clé* à gauche de cet élément, supprimant donc l'inversion. Comme la ligne 5 ne déplace que les éléments qui sont inférieurs à *clé*, elle ne déplace que des éléments qui correspondent à des inversions. En d'autres termes, chaque itération de la boucle **tant que** des lignes 5–7 correspond à l'élimination d'une inversion.

d. Suivant l'indication, nous modifions le tri par fusion de façon à compter le nombre d'inversions en temps $\Theta(n \lg n)$.

Pour commencer, définissons une ***fusion-inversion*** comme étant une situation, à l'intérieur

de l'exécution du tri par fusion, où la procédure FUSION, après avoir copié $A[p .. q]$ dans L et $A[q + 1 .. r]$ dans R, a des valeurs x dans L et y dans R telles que $x > y$. Considérons une inversion (i, j), et soient $x = A[i]$ et $y = A[j]$ tels que $i < j$ et $x > y$. Nous affirmons que, si nous devions exécuter le tri par fusion, il y aurait une et une seule fusion-inversion impliquant x et y. Pour voir pourquoi, observez que le seul moyen permettant aux éléments du tableau de modifier leurs positions se trouve dans la procédure FUSION. En outre, comme FUSION conserve les éléments de L dans le même ordre relatif qu'ils ont les uns par rapport aux autres, et pareillement pour R, le seul moyen permettant à deux éléments de modifier leur ordre mutuel est que le plus grand apparaisse dans L et le plus petit dans R. Par conséquent, il y a au moins une fusion-inversion impliquant x et y. Pour voir pourquoi il n'y en a qu'une, observez que, après un appel à FUSION qui implique x et y, ils sont dans le même sous-tableau trié ; ils apparaissent donc tous les deux dans L, ou tous les deux dans R, dans tout appel subséquent. Nous avons donc prouvé notre affirmation.

Nous avons montré que chaque inversion implique une fusion-inversion. En fait, la correspondance entre inversions et fusions-inversions est bijective. Supposons que nous ayons une fusion-inversion impliquant les valeurs x et y, où initialement x était $A[i]$ et y $A[j]$. Comme nous avons une fusion-inversion, $x > y$. Et comme x est dans L et y dans R, x doit être dans un sous-tableau qui précède le sous-tableau contenant y. Par conséquent, x a commencé à une position i précédant la position originale j de y, ce qui fait que (i, j) est une inversion.

Ayant prouvé la bijection entre inversions et fusions-inversions, il nous suffit de compter les fusions-inversions.

Considérons une fusion-inversion impliquant y dans R. Soit z la plus petite valeur de L qui est supérieure à y. À un certain stade du processus de fusion, z et y sont les valeurs « exposées » dans L et R, c'est-à-dire que nous avons $z = L[i]$ et $y = R[j]$ en ligne 13 de FUSION. À cet instant il y a des fusions-inversions impliquant y et $L[i]$, $L[i + 1]$, $L[i + 2]$,…, $L[n_1]$, et ces $n_1 - i + 1$ fusions-inversions sont les seules impliquant y. Par conséquent, nous devons détecter la première fois que z et y deviennent exposées pendant la procédure FUSION et ajouter à notre compte total de fusions-inversions la valeur de $n_1 - i + 1$ à cet instant.

Le pseudo-code suivant, modélisé d'après le tri par fusion, fonctionne comme précédemment expliqué. Il trie aussi le tableau A.

```
COMPTE-INVERSIONS(A, p, r)
inversions = 0
si p < r
    q = ⌊(p + r)/2⌋
    inversions = inversions + COMPTE-INVERSIONS(A, p, q)
    inversions = inversions + COMPTE-INVERSIONS(A, q + 1, r)
    inversions = inversions + FUSIONS-INVERSIONS(A, p, q, r)
retourner inversions
FUSIONS-INVERSIONS(A, p, q, r)
n1 = q – p + 1
n2 = r – q
soient L[1 .. n1] et R[1 .. n2] de nouveaux tableaux
pour i = 1 à n1
    L[i] = A[p + i – 1]
pour j = 1 à n2
    R[j] = A[q + j]
L[n1 +1] = ∞
```

```
R[n2 + 1] = ∞
i = 1
j = 1
inversions = 0
compté = FAUX
pour k = p à r
    si compté == FAUX et R[j] < L[i]
        inversions = inversions + n1 – i + 1
        compté = VRAI
    si L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    sinon A[k] = R[j]
        j = j + 1
        compté = FAUX
retourner inversions
```

L'appel initial est COMPTE-INVERSIONS(A, 1, n).

Dans FUSIONS-INVERSIONS, la variable booléenne *compté* indique si nous avons compté les fusions-inversions impliquant $R[j]$. Nous les comptons la première fois que $R[j]$ est exposé et qu'en même temps une valeur supérieure à $R[j]$ devient exposée dans le tableau L. Nous réglons *compté* sur FAUX chaque fois qu'une nouvelle valeur devient exposée dans R. Nous n'avons pas besoin de nous soucier des fusions-inversions impliquant la sentinelle ∞ dans R, car aucune valeur de L ne dépassera ∞.

Comme nous n'avons ajouté qu'une quantité constante de travail supplémentaire à chaque appel de procédure et à chaque itération de la dernière boucle **pour** de la procédure de fusion, le temps total d'exécution du pseudo-code précédent est le même que pour le tri par fusion : $\Theta(n \lg n)$.

Solutions choisies du chapitre 3 : Croissance des fonctions

Solution de l'exercice 3.1-2

Pour montrer que $(n + a)^b = \Theta(n^b)$, nous devons trouver des constantes c_1, c_2, $n_0 > 0$ telles que $0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b$ pour tout $n \geq n_0$.

Notez que

$$\begin{aligned} n + a &\leq n + |a| \\ &\leq 2n \qquad \text{quand } |a| \leq n, \end{aligned}$$

et

$$\begin{aligned} n + a &\geq n - |a| \\ &\geq \frac{1}{2} n \qquad \text{quand } |a| \leq \frac{1}{2} n. \end{aligned}$$

Donc, quand $n \geq 2|a|$,

$$0 \leq \frac{1}{2} n \leq n + a \leq 2n.$$

Comme $b > 0$, l'inégalité reste vraie quand toutes les parties sont élevées à la puissance b :

$$0 \leq (\frac{1}{2} n)^b \leq (n + a)^b \leq (2n)^b,$$

$$0 \leq (\frac{1}{2})^b n^b \leq (n + a)^b \leq 2^b n^b.$$

Donc, $c_1 = (1/2)^b$, $c_2 = 2^b$ et $n_0 = 2|a|$ satisfont à la définition.

Solution de l'exercice 3.1-3

Soit $T(n)$ le temps d'exécution. $T(n) \geq O(n^2)$ signifie que $T(n) \geq f(n)$ pour une certaine fonction $f(n)$ de l'ensemble $O(n^2)$. Cette assertion reste vraie pour n'importe quel temps d'exécution $T(n)$, puisque la fonction $g(n) = 0$ pour tout n est dans $O(n^2)$ et que les temps d'exécution sont toujours positifs. Par conséquent, l'assertion ne nous dit rien sur le temps d'exécution.

Solution de l'exercice 3.1-4

$2^{n+1} = O(2^n)$, mais $2^{2n} \neq O(2^n)$.

Pour montrer que $2^{n+1} = O(2^n)$, nous devons trouver des constantes c, $n_0 > 0$ telles que $0 \leq 2^{n+1} \leq c \cdot 2^n$ pour tout $n \geq n_0$.

Comme $2^{n+1} = 2 \cdot 2^n$ pour tout n, nous pouvons satisfaire à la définition avec $c = 2$ et $n_0 = 1$.

Pour montrer que $2^{2n} \neq O(2^n)$, supposons qu'il existe des constantes c, $n_0 > 0$ telles que $0 \leq 2^{2n} \leq c \cdot 2^n$ pour tout $n \geq n_0$.

Alors $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. Mais aucune constante n'est plus supérieure à tous les 2^n, de sorte que l'hypothèse entraîne une contradiction.

Solution de l'exercice 3.2-4

$\lceil \lg n \rceil !$ n'est pas borné polynomialement, mais $\lceil \lg \lg n \rceil !$ l'est.

Démontrer qu'une fonction $f(n)$ est bornée polynomialement équivaut à démontrer que $\lg(f(n)) = O(\lg n)$ pour les raisons suivantes.

- Si f est bornée polynomialement, il existe des constants c, k, n_0 telles que, pour tout $n \geq n_0$, $f(n) \leq cn^k$. Par conséquent, $\lg(f(n)) \leq kc \lg n$, ce qui signifie, puisque c et k sont constants, que $\lg(f(n)) = O(\lg n)$.
- De même, si $\lg(f(n)) = O(\lg n)$, alors f est bornée polynomialement.

Dans les démonstrations, nous utiliserons les deux faits suivants :

1 $\lg(n!) = \Theta(n \lg n)$ (d'après l'équation (3.19)).
2 $\lceil \lg n \rceil = \Theta(\lg n)$, car
   - $\lceil \lg n \rceil \geq \lg n$
   - $\lceil \lg n \rceil < \lg n + 1 \leq 2 \lg n$ pour tout $n \geq 2$

$$\lg(\lceil \lg n \rceil !) = \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil)$$
$$= \Theta(\lg n \lg \lg n)$$
$$= \omega(\lg n).$$

Par conséquent, $\lg(\lceil \lg n \rceil !) \neq O(\lg n)$ et donc $\lceil \lg n \rceil !$ n'est pas borné polynomialement.

$$\begin{aligned}
\lg(\lceil \lg \lg n \rceil !) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\
&= \Theta(\lg \lg n \lg \lg \lg n) \\
&= o((\lg \lg n)^2) \\
&= o(\lg^2(\lg n)) \\
&= o(\lg n).
\end{aligned}$$

La dernière étape précédente découle de la propriété selon laquelle une fonction polylogarithmique croît plus lentement que toute fonction polynomiale positive ; c'est-à-dire que, pour des constantes a, $b > 0$, nous avons $\lg^b n = o(n^a)$. En substituant $\lg n$ à n, 2 à b et 1 à a, on obtient $\lg^2(\lg n) = o(\lg n)$.

Par conséquent, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$ et donc $\lceil \lg \lg n \rceil !$ est borné polynomialement.

Solutions choisies du chapitre 4 : Diviser pour régner

Solution de l'exercice 4.2-4

Si vous pouvez multiplier des matrices 3 x 3 à l'aide de k multiplications, alors vous pouvez multiplier des matrices $n \times n$ en multipliant récursivement des matrices $n/3 \times n/3$, en temps $T(n) = kT(n/3) + \Theta(n^2)$.

En utilisant la méthode générale pour résoudre cette récurrence, considérons le rapport entre $n^{\log_3 k}$ et n^2 :

- Si $\log_3 k = 2$, c'est le cas 2 qui s'applique et $T(n) = \Theta(n^2 \lg n)$. Ici, $k = 9$ et $T(n) = o(n^{\lg 7})$.
- Si $\log_3 k < 2$, c'est le cas 3 qui s'applique et $T(n) = \Theta(n^2)$. Ici, $k < 9$ et $T(n) = o(n^{\lg 7})$.
- Si $\log_3 k > 2$, c'est le cas 1 qui s'applique et $T(n) = \Theta(n^{\log_3 k})$. Ici, $k > 9$. $T(n) = o(n^{\lg 7})$ quand $\log_3 k < \lg 7$, c'est-à-dire quand $k < 3^{\lg 7} \approx 21{,}85$. Le plus grand entier k correspondant est 21.

Donc, $k = 21$ et la durée d'exécution est $\Theta(n^{\log_3 k}) = \Theta(n^{\log_3 21}) = O(n^{2{,}80})$ (car $\log_3 21 \approx 2{,}77$).

Solution de l'exercice 4.4-6

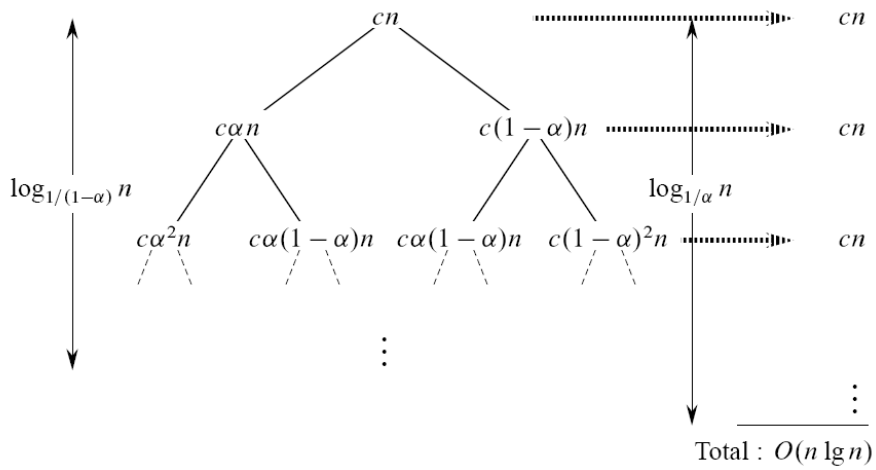
Le plus court chemin entre la racine et une feuille de l'arbre récursif est $n \to (1/3)n \to (1/3)^2 n \to \ldots \to 1$. Comme $(1/3)^k n = 1$ quand $k = \log_3 n$, la hauteur de la partie de l'arbre où chaque nœud a deux enfants est $\log_3 n$. Comme les valeurs à chacun de ces niveaux de l'arbre font en tout cn, la solution de la récurrence est au moins $cn \log_3 n = \Omega(n \lg n)$.

Solution de l'exercice 4.4-9

$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$

Nous avons vu la solution de la récurrence $T(n) = T(n/3) + T(2n/3) + cn$ dans le texte. La récurrence présente se résout de façon similaire.

Sans nuire à la généralité, soit $\alpha \geq 1 - \alpha$, de sorte que $0 < 1 - \alpha \leq 1/2$ et $1/2 \leq \alpha < 1$.



L'arbre récursif est plein pour $\log_{1/(1-\alpha)} n$ niveaux, dont chacun contribue pour cn, de sorte que nous supputons $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$. Il a $\log_{1/\alpha} n$ niveaux, dont chacun contribue pour $\leq cn$, de sorte que nous supputons $O(n \log_{1/\alpha} n) = O(n \lg n)$.
Montrons maintenant que $T(n) = \Theta(n \lg n)$ par substitution. Pour prouver le majorant, nous devons montrer que $T(n) \leq dn \lg n$ pour une constante $d > 0$ convenable.

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\ &\leq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\ &= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\ &= dn \lg n + dn\,(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\ &\leq dn \lg n, \end{aligned}$$

si $dn\,(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \leq 0$. Cette condition équivaut à

$$d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \leq -c.$$

Comme $1/2 \leq \alpha < 1$ et $0 < 1 - \alpha \leq 1/2$, nous avons $\lg \alpha < 0$ et $\lg(1-\alpha) < 0$. Donc, $\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha) < 0$, de sorte que, quand nous multiplions les deux membres de l'inégalité par ce facteur, nous devons inverser le sens :

$$d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$$

ou

$$d \geq \frac{-c}{-\alpha \lg \alpha + -(1-\alpha)\lg(1-\alpha)}$$

La fraction à droite étant une constante positive, il suffit de prendre n'importe quelle valeur de d qui soit supérieure ou égale à cette fraction.
Pour prouver le minorant, nous devons montrer que $T(n) \geq dn \lg n$ pour une constante $d > 0$ convenable. Nous pouvons employer la même démonstration que pour le majorant, en substituant $\geq$ à $\leq$, et nous obtenons la condition

$$0 < d \leq \frac{-c}{-\alpha \lg \alpha - (1-\alpha)\lg(1-\alpha)}$$

Par conséquent, $T(n) = \Theta(n \lg n)$.

Solutions choisies du chapitre 5 : Analyse probabiliste et algorithmes randomisés

Solution de l'exercice 5.2-1

Comme EMBAUCHE-SECRÉTAIRE embauche toujours la candidate 1, elle embauche une et une seule fois si l'on n'embauche pas d'autres candidates que la candidate 1. Cet événement se produit quand la candidate 1 est la meilleure candidate des n, ce qui se produit avec la probabilité $1/n$.

EMBAUCHE-SECRÉTAIRE embauche n fois si chaque candidate est meilleure que toutes celles vues (et embauchées) auparavant. Cet événement se produit précisément quand la liste d'évaluations passée à l'algorithme est $\langle 1, 2, \ldots, n \rangle$, ce qui se produit avec la probabilité $1/n!$.

Solution de l'exercice 5.2-4

Une autre façon de considérer le problème du vestiaire à chapeaux consiste à dire que nous voulons déterminer le nombre espéré de points fixes dans une permutation aléatoire. (Un ***point fixe*** d'une permutation π est une valeur i pour laquelle $\pi(i) = i$.) Nous pourrions énumérer les $n!$ permutations, compter le nombre total de points fixes, puis diviser par $n!$ pour déterminer le nombre moyen de points fixes par permutation. Ce serait un processus laborieux et la réponse s'avérerait être 1. Nous pouvons, cependant, employer des variables aléatoires indicatrices pour arriver plus facilement à la même réponse.

Définissons une variable aléatoire X qui est égale au nombre de clients qui récupèrent leur chapeau personnel, de sorte que nous devons calculer $E[X]$.

Pour $i = 1, 2, \ldots, n$, définissons la variable aléatoire indicatrice

$X_i = I\{\text{client } i \text{ récupère son chapeau personnel}\}$.

Alors $X = X_1 + X_2 + \ldots + X_n$.

L'ordre des chapeaux étant aléatoire, chaque client a une probabilité $1/n$ de récupérer son chapeau personnel. Autrement dit, $\Pr\{X_i = 1\} = 1/n$, ce qui, d'après le lemme 5.1, implique que $E[X_i] = 1/n$.

Par conséquent,

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \quad (\textit{linéarité de l'espérance}) \\ &= \sum_{i=1}^{n} E[X_i] \\ &= 1 \end{aligned}$$

et donc nous espérons qu'un client et un seul récupère son chapeau personnel.
Notez que c'est un cas où les variables aléatoires indicatrices ne sont *pas* indépendantes. Par exemple, si $n = 2$ et $X_1 = 1$, alors X_2 doit aussi être égal à 1. Inversement, si $n = 2$ et $X_1 = 0$, alors X_2 doit aussi être égal à 0. Malgré la dépendance, $\Pr\{X_i = 1\} = 1/n$ pour tout i et la linéarité de l'espérance reste vraie. Par conséquent, nous pouvons employer la technique des variables aléatoires indicatrices même en cas de dépendance.

Solution de l'exercice 5.2-5

Soit X_{ij} une variable aléatoire indicatrice pour l'événement suivant : la paire $A[i]$, $A[j]$ pour $i < j$ est inversée, c'est-à-dire que $A[i] > A[j]$. Plus précisément, définissons $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ pour $1 \le i < j \le n$. Nous avons $\Pr\{X_{ij} = 1\} = 1/2$, car, étant donnés deux nombres aléatoires distincts, la probabilité que le premier soit supérieur au second est 1/2. D'après le lemme 5.1, $\mathrm{E}[X_{ij}] = 1/2$.

Soit X la variable aléatoire indiquant le nombre total de paires inversées dans le tableau, de sorte que

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Comme nous voulons le nombre espéré de paires inversées, nous prenons l'espérance des deux côtés de l'équation précédente pour obtenir

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

La linéarité de l'espérance donne

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} \\ &= \binom{n}{2} \frac{1}{2} \\ &= \frac{n(n-1)}{2} \cdot \frac{1}{2} \\ &= \frac{n(n-1)}{4} \end{aligned}$$

Par conséquent, le nombre espéré de paires inversées est $n(n-1)/4$.

Solution de l'exercice 5.3-2

Bien que PERMUTE-SANS-IDENTITÉ ne produise pas la permutation identité, il y a d'autres permutations qu'elle ne réussit pas à produire. Par exemple, considérons son fonctionnement quand $n = 3$, quand elle devrait pouvoir produire les $n! - 1 = 5$ permutations autres que la permutation identité. La boucle **pour** itère pour $i = 1$ et $i = 2$. Quand $i = 1$, l'appel à RANDOM retourne l'une des deux valeurs possibles (2 ou 3) ; quand $i = 2$, l'appel à RANDOM ne retourne qu'une seule valeur (3). Donc, PERMUTE-SANS-IDENTITÉ ne peut produire que $2 \cdot 1 = 2$ permutations possibles, au lieu des 5 requises.

Solution de l'exercice 5.3-4

PERMUTE-PAR-CYCLE choisit *offset* en tant qu'entier aléatoire dans la plage $1 \leq offset \leq n$, puis effectue une rotation cyclique du tableau. C'est-à-dire que $B[((i + offset - 1) \bmod n) + 1] = A[i]$ pour $i = 1, 2, \ldots, n$. (La soustraction et l'addition de 1 dans le calcul des indices viennent de ce que l'indexation commence à 1. Si l'indexation commençait à 0, le calcul des indices aurait été plus simple : $B[(i + offset) \bmod n] = A[i]$ pour $i = 0, 1, \ldots, n - 1$.)

Donc, une fois que *offset* est déterminé, toute la permutation l'est aussi. Comme chaque valeur de *offset* se produit avec la probabilité $1/n$, chaque élément $A[i]$ a la probabilité $1/n$ de finir à la position $B[j]$.

Cette procédure ne produit pas de permutation aléatoire uniforme, en revanche, car elle ne peut produire que n permutations différentes. Donc, n permutations se produisent avec la probabilité $1/n$ et les $n! - n$ permutations restantes se produisent avec la probabilité 0.

Solutions choisies du chapitre 6 : Tri par tas

Solution de l'exercice 6.1-1

Étant un arbre binaire presque complet (complet à tous les niveaux, sauf peut-être au plus bas), un tas contient au plus 2^{h+1} éléments (s'il est complet) et au moins $2^h - 1 + 1 = 2^h$ éléments (si le niveau le plus bas n'a que 1 élément, alors que les autres niveaux sont complets).

Solution de l'exercice 6.1-2

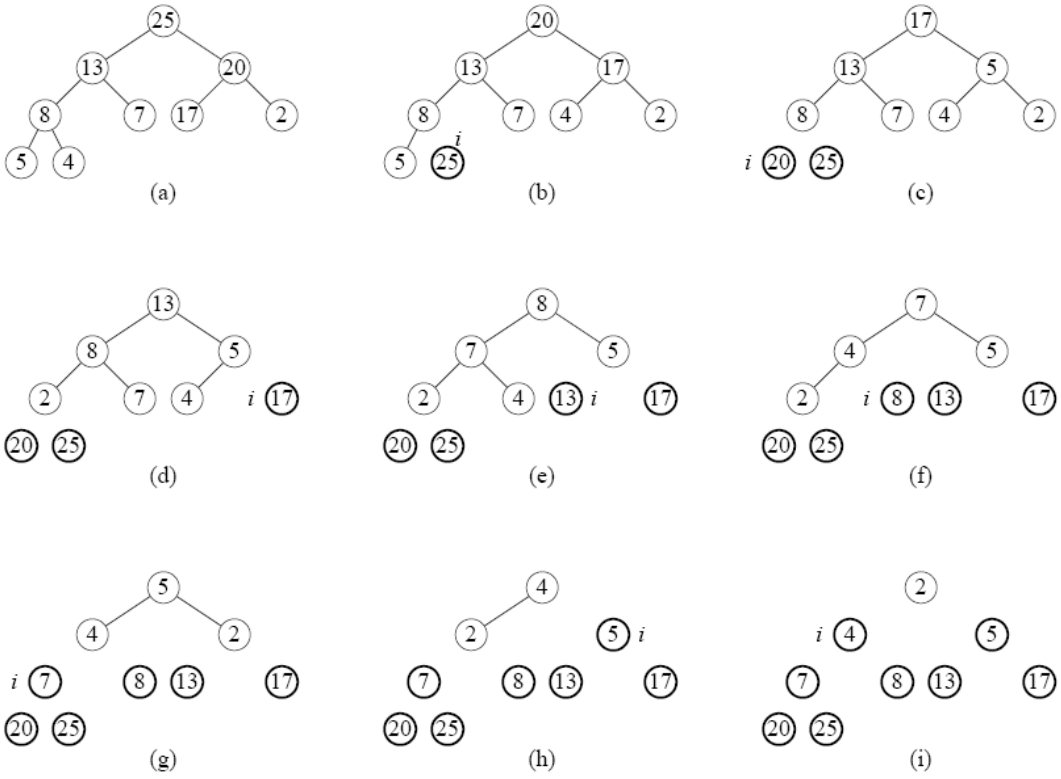
Étant donné un tas à n éléments de hauteur h, l'exercice 6.1-1 montre que
$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$

Par conséquent, $h \leq \lg n < h + 1$. Comme h est un entier, $h = \lfloor \lg n \rfloor$ (par définition de $\lfloor \ \rfloor$).

Solution de l'exercice 6.2-6

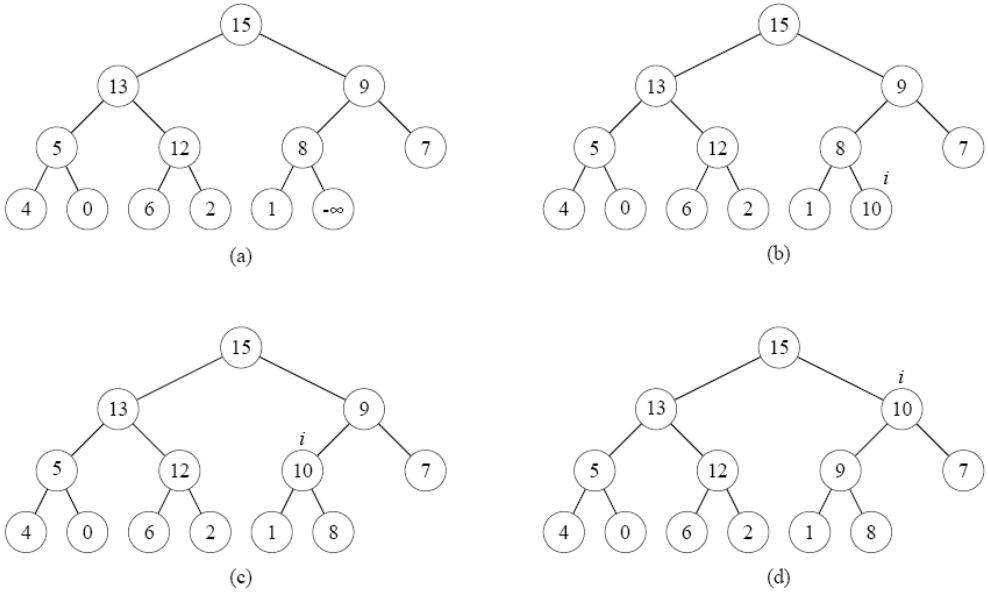
Si vous placez dans la racine une valeur qui est inférieure à toutes les autres valeurs des sous-arbres de gauche et de droite, alors il y aura appel récursif de ENTASSER-MAX jusqu'à ce qu'une feuille soit atteinte. Pour que les appels récursifs traversent le plus long chemin menant à une feuille, choisissez des valeurs telles que ENTASSER-MAX fasse toujours de la récursivité sur l'enfant de gauche. La procédure suit la branche de gauche quand l'enfant de gauche est supérieur ou égal à l'enfant de droite, de sorte qu'en mettant 0 dans la racine et 1 dans tous les autres nœuds, par exemple, vous y arriverez. Avec de telles valeurs, ENTASSER-MAX sera appelé h fois (où h est la hauteur du tas, c'est-à-dire le nombre d'arcs du plus long chemin reliant la racine à une feuille), et donc son temps d'exécution sera $\Theta(h)$ (puisque chaque appel effectue un travail $\Theta(1)$), soit $\Theta(\lg n)$. Comme nous avons un cas où le temps d'exécution de ENTASSER-MAX vaut $\Theta(\lg n)$, son temps d'exécution dans le cas le plus défavorable est $\Omega(\lg n)$.

Solution de l'exercice 6.4-1



A	2	4	5	7	8	13	17	20	25

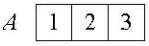
Solution de l'exercice 6.5-2



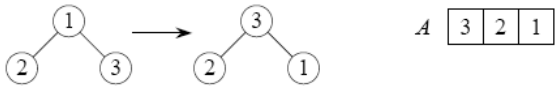
Solution du problème 6-1

a. Les procédures CONSTRUIRE-TAS-MAX et CONSTRUIRE-TAS-MAX' ne créent pas toujours le même tas à partir du même tableau passé en entrée. Considérons le contre-exemple suivant.

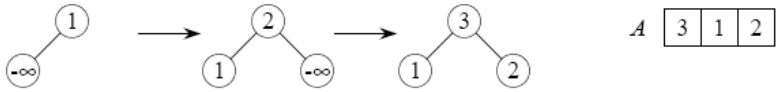
Tableau en entrée A :



CONSTRUIRE-TAS-MAX(*A*) :



CONSTRUIRE-TAS-MAX'(*A*) :



b. Une borne supérieure temporelle $O(n \lg n)$ découle immédiatement de ce qu'il y a $n - 1$ appels à INSÉRER-TAS-MAX, chacun prenant un temps $O(\lg n)$. Pour une borne inférieure $\Omega(n \lg n)$, considérons le cas où le tableau en entrée est donné dans l'ordre strictement croissant. Chaque appel à INSÉRER-TAS-MAX force AUGMENTER-CLÉ-TAS à remonter jusqu'à la racine. Comme la profondeur du nœud i est $\lfloor \lg i \rfloor$, le temps total est

$$\begin{aligned}
\sum_{i=1}^{n} \Theta\left(\lfloor \lg i \rfloor\right) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\left\lfloor \lg \left\lceil n/2 \right\rceil \right\rfloor\right) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\left\lfloor \lg (n/2) \right\rfloor\right) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\lfloor \lg n - 1 \rfloor\right) \\
&= n/2 \cdot \Theta(\lg n) \\
&= \Omega(n \lg n)
\end{aligned}$$

Par conséquent, dans le cas le plus défavorable, CONSTRUIRE-TAS-MAX' exige un temps $\Theta(n \lg n)$ pour construire un tas à n éléments.

Solutions choisies du chapitre 7 : Tri rapide

Solution de l'exercice 7.2-3

PARTITION effectue un « partitionnement dans le cas le plus défavorable » quand les éléments sont en ordre décroissant. La procédure diminue la taille du sous-tableau concerné de seulement 1 à chaque étape, ce qui donne, comme nous l'avons vu, une durée d'exécution $\Theta(n^2)$.

En particulier, PARTITION, étant donné un sous-tableau $A[p \,..\, r]$ d'éléments distincts en ordre décroissant, produit une partition vide dans $A[p \,..\, q - 1]$, place le pivot (originellement en $A[r]$) dans $A[p]$ et produit une partition $A[p + 1 \,..\, r]$ contenant un seul élément de moins que $A[p \,..\, r]$. La récurrence pour TRI-RAPIDE devient $T(n) = T(n - 1) + \Theta(n)$, dont la solution est $T(n) = \Theta(n^2)$.

Solution de l'exercice 7.2-5

La profondeur minimum suit un chemin qui prend toujours la plus petite partie de la partition, c'est-à-dire qui multiplie le nombre d'éléments par α. Une itération fait tomber le nombre d'éléments de n à αn et i itérations font tomber le nombre d'éléments à $\alpha^i n$. Dans une feuille, il reste juste un élément ; et donc, à une feuille de profondeur minimale ayant la profondeur m, nous avons $\alpha^m n = 1$. Donc, $\alpha^m = 1/n$. En prenant les logarithmes, nous obtenons $m \lg \alpha = -\lg n$, soit $m = -\lg n / \lg \alpha$.

De même, la profondeur maximum revient à prendre toujours la plus grande partie de la partition, c'est-à-dire à conserver chaque fois une fraction $1 - \alpha$ des éléments. La profondeur maximum M est atteinte quand il ne reste qu'un seul élément, c'est-à-dire quand $(1 - \alpha)^M n = 1$. Donc, $M = -\lg n / \lg(1 - \alpha)$.

Toutes ces équations sont approximatives, car nous ignorons les arrondis aux parties entières inférieures ou supérieures.

Solutions choisies du chapitre 8 : Tri en temps linéaire

Solution de l'exercice 8.1-3

Si le tri est exécuté en temps linéaire pour m permutations en entrée, alors la hauteur h de la portion de l'arbre de décision composée des m feuilles correspondantes et de leurs ancêtres est linéaire.

Nous employons la même démonstration que celle du théorème 8.1 pour montrer que c'est impossible pour $m = n!/2$ ou $n!/2^n$.

Nous avons $2^h \geq m$, ce qui nous donne $h \geq \lg m$. Pour toutes les valeurs possibles de m données ici, $\lg m = \Omega(n \lg n)$, d'où $h = \Omega(n \lg n)$.

En particulier,

$$\lg \frac{n!}{2} = \lg n! \geq n \lg n - n \lg e - 1,$$

$$\lg \frac{n!}{n} = \lg n! - \lg n \geq n \lg n - n \lg e - \lg n,$$

$$\lg \frac{n!}{2^n} = \lg n! - n \geq n \lg n - n \lg e - n.$$

Solution de l'exercice 8.2-3

La solution suivante répond aussi à l'exercice 8.2-2.

Remarquez que la démonstration de validité dans le texte ne dépend pas de l'ordre dans lequel est traité A. L'algorithme est correct, quel que soit l'ordre employé !

Mais l'algorithme modifié n'est pas stable. Comme précédemment, dans la boucle **pour** finale, un élément égal à un élément pris dans A précédemment est placé avant l'élément antérieur (c'est-à-dire, dans une position d'index inférieure) dans le tableau en sortie B. L'algorithme original était stable, car un élément pris dans A ultérieurement démarrait à un indice inférieur à un élément pris précédemment. Mais dans l'algorithme modifié, un élément pris dans A ultérieurement démarre à un indice supérieur à un élément pris précédemment.

En particulier, l'algorithme place encore les éléments ayant la valeur k dans les emplacements $C[k-1] + 1$ à $C[k]$, mais dans l'ordre inverse de leur apparition dans A.

Solution de l'exercice 8.3-3

Cas de base : Si $d = 1$, il n'y a qu'un seul chiffre, de sorte qu'un tri sur ce chiffre trie le tableau.

Étape de récurrence (inductive) : Supposons que le tri par base fonctionne pour $d - 1$ chiffres et montrons qu'il fonctionne pour d chiffres.

Le tri par base trie séparément sur chaque chiffre, en commençant au chiffre 1. Donc, le tri par base de d chiffres, qui trie sur les chiffres 1, …, d, équivaut au tri par base des $d - 1$ chiffres de poids faible, suivi d'un tri sur le chiffre d. D'après notre hypothèse de récurrence, le tri des $d - 1$ chiffres de poids faible fonctionne, de sorte que juste avant le tri sur le chiffre d, les éléments sont triés selon leurs $d - 1$ chiffres de poids faible.

Le tri sur le chiffre d trie les éléments en fonction de leur d-ième chiffre. Considérons deux éléments, a et b, ayant pour d-ième chiffre a_d et b_d respectivement.

- Si $a_d < b_d$, le tri placera a avant b, ce qui est correct, puisque $a < b$ indépendamment des chiffres de poids faible.
- Si $a_d > b_d$, le tri placera a après b, ce qui est correct, puisque $a > b$ indépendamment des chiffres de poids faible.
- Si $a_d = b_d$, le tri laissera a et b dans l'ordre qu'ils avaient, car il est stable. Mais cet ordre est déjà correct, vu que l'ordre correct de a et b est déterminé par les $d - 1$ chiffres de poids faible quand les d-ièmes chiffres sont égaux et que les éléments sont déjà triés selon leurs $d - 1$ chiffres de poids faible.

Si le tri intermédiaire n'était pas stable, il risquerait de réorganiser les éléments dont les d-ièmes chiffres sont égaux (éléments qui *étaient* dans le bon ordre après le tri sur leurs chiffres de poids faible).

Solution de l'exercice 8.3-4

Traitez les nombres comme des nombres de 3 chiffres en base n. Chaque chiffre va de 0 à $n - 1$. Triez ces nombres de 3 chiffres à l'aide du tri par base.

Comme il y a 3 appels au tri par dénombrement, chacun prenant un temps $\Theta(n + n) = \Theta(n)$, la durée totale est $\Theta(n)$.

Solution du problème 8-1

a. Pour qu'un algorithme de comparaison A trie, deux permutations en entrée ne peuvent pas atteindre la même feuille de l'arbre de décision, de sorte qu'il doit y avoir au moins $n!$ feuilles atteintes dans T_A, une pour chaque permutation en entrée possible. Comme A est un algorithme déterministe, il doit toujours atteindre la même feuille quand on lui passe en entrée une permutation particulière, de sorte qu'il y a au plus $n!$ feuilles atteintes (une pour chaque permutation). Par conséquent, il y a exactement $n!$ feuilles atteintes, une pour chaque permutation en entrée.

Ces $n!$ feuilles ont chacune la probabilité $1/n!$, car chacune des $n!$ permutations possibles est l'entrée ayant la probabilité $1/n!$. Toutes les autres feuilles ont la probabilité 0, car elles ne sont atteintes pour aucune entrée.

Sans nuire à la généralité, nous pouvons supposer pour le reste du problème que les chemins menant uniquement à des feuilles de probabilité 0 ne sont pas dans l'arbre, car ils ne peuvent pas affecter la durée d'exécution du tri. Autrement dit, nous pouvons supposer que T_A se compose uniquement des $n!$ feuilles étiquetées $1/n!$ et de leurs ancêtres.

b. Si $k > 1$, alors la racine de T n'est pas une feuille. Cela implique que toutes les feuilles de T sont des feuilles de LT et RT. Comme chaque feuille ayant la profondeur h dans LT ou RT a la profondeur $h + 1$ dans T, $D(T)$ est forcément la somme de $D(LT)$, $D(RT)$ et k, nombre total de feuilles. Pour démontrer cette dernière assertion, soit $d_T(x)$ = profondeur du nœud x dans l'arbre T.

Alors,

$$\begin{aligned} D(T) &= \sum_{x \in \text{feuilles}(T)} d_T(x) \\ &= \sum_{x \in \text{feuilles}(LT)} d_T(x) + \sum_{x \in \text{feuilles}(RT)} d_T(x) \\ &= \sum_{x \in \text{feuilles}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{feuilles}(RT)} (d_{RT}(x) + 1) \\ &= \sum_{x \in \text{feuilles}(LT)} d_{LT}(x) + \sum_{x \in \text{feuilles}(RT)} d_{RT}(x) + \sum_{x \in \text{feuilles}(T)} 1 \end{aligned}$$

c. Pour montrer que $d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k - i) + k\}$, nous montrerons séparément que

$d(k) \le \min_{1 \le i \le k-1}\{d(i) + d(k - i) + k\}$

et que

$d(k) \ge \min_{1 \le i \le k-1}\{d(i) + d(k - i) + k\}$

- Pour montrer que $d(k) \le \min_{1 \le i \le k-1}\{d(i) + d(k - i) + k\}$, il suffit de montrer que $d(k) \le d(i) + d(k - i) + k$, pour $i = 1, 2, \ldots, k - 1$. Pour tout i entre 1 et $k - 1$, nous pouvons trouver des arbres RT à i feuilles et LT à $k - i$ feuilles tels que $D(RT) = d(i)$ et $D(LT) = d(k - i)$. Construisons T de telle façon que RT et LT soient les sous-arbres gauche et droit de la racine de T. Alors

  $d(k) \le D(T)$ (d'après la définition de d comme valeur de $D(T)$ min)
  $= D(RT) + D(LT) + k$ (d'après la partie (b))
  $= d(i) + d(k - i) + \text{k}$ (d'après le choix de RT et LT)

- Pour montrer que $d(k) \ge \min_{1 \le i \le k-1}\{d(i) + d(k - i) + k\}$, il suffit de montrer que $d(k) \ge d(i) + d(k - i) + k$, pour un certain i dans $\{1, 2, \ldots, k - 1\}$. Prenons l'arbre T à k feuilles tel que $D(T) = d(k)$, soient RT et LT les sous-arbres gauche et droit de T, et soit i le nombre de feuilles de RT. Alors, $k - i$ est le nombre de feuilles de LT et

  $d(k) = D(T)$ (d'après le choix de T)
  $= D(RT) + D(LT) + k$ (d'après la partie (b))
  $\ge d(i) + d(k - i) + \text{k}$ (d'après la définition de d comme valeur de $D(T)$ min)

  Ni i ni $k - i$ ne peuvent valoir 0 (et donc $1 \le i \le k - 1$) ; en effet, si l'une de ces valeurs était 0, RT ou LT contiendrait toutes les k feuilles de T et ce sous-arbre à k feuilles aurait un D égal à $D(T) - k$ (d'après la partie (b)), ce qui contredirait le choix de T comme étant l'arbre à k feuilles qui a le D minimum.

d. Soit $f_k(i) = i \lg i + (k - i) \lg(k - i)$. Pour trouver la valeur de i qui minimise f_k, trouvons le i pour lequel la dérivée de f_k par rapport à i vau 0 :

$$\begin{aligned} f_k'(i) &= \frac{d}{di}\left(\frac{i\ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{\ln i + 1 - \ln(k-i) - 1}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \end{aligned}$$

vaut 0 en $i = k/2$. Pour vérifier qu'il s'agit bien d'un minimum (et non d'un maximum), vérifions que la dérivée seconde de f_k est positive en $i = k/2$:

$$\begin{aligned} f_k''(i) &= \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right) \\ &= \frac{1}{\ln 2}\left(\frac{1}{i} + \frac{1}{k-i}\right) \end{aligned}$$

$$\begin{aligned} f_k''(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k} + \frac{2}{k}\right) \\ &= \frac{1}{\ln 2} \cdot \frac{4}{k} \\ &> 0 \quad \text{puisque k > 1.} \end{aligned}$$

Nous allons utiliser la substitution pour montrer que $d(k) = \Omega(k \lg k)$. Le cas de base de la récurrence est satisfait, car $d(1) \geq 0 = c \cdot 1 \cdot \lg 1$ pour toute constante c. Pour l'étape de récurrence, supposons que $d(i) \geq ci \lg i$ pour $1 \leq i \leq k - 1$, où c est une certaine constante à déterminer.

$$\begin{aligned} d(k) &= \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\} \\ &\geq \min_{1 \leq i \leq k-1} \{c\ i \lg i + (k-i)\lg(k-i)\ + k\} \\ &= \min_{1 \leq i \leq k-1} \{c f_k(i) + k\} \\ &= c\left(\frac{k}{2}\lg\frac{k}{2}\left(k - \frac{k}{2}\right)\lg\left(k - \frac{k}{2}\right)\right) + k \\ &= ck \lg\left(\frac{k}{2}\right) + k \\ &= c(k \lg k - k) + k \\ &= ck \lg k + (k - ck) \\ &\geq ck \lg k \qquad \text{si c} \leq 1 \end{aligned}$$

et donc $d(k) = \Omega(k \lg k)$.

e. En combinant le résultat de la partie (d) et le fait que T_A (tel que modifié dans notre solution de la partie (a)) a $n!$ feuilles, nous pouvons conclure que

$D(T_{A)} \geq d(n!) = \Omega(n! \lg(n!))$.

$D(T_{A)}$ est la somme des longueurs de chemin { }d'arbre de décision pour trier toutes les permutations en entrée et les longueurs de chemin sont proportionnelles à la durée d'exécution. Comme les $n!$ permutations ont une probabilité égale valant $1/n!$, la durée espérée pour trier n éléments aléatoires (1 permutation en entrée) est la durée totale pour

toutes les permutations divisée par $n!$:

$$\frac{\Omega(n!\lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n)$$

f. Nous allons montrer comment modifier un arbre (algorithme) de décision randomisé pour définir un arbre (algorithme) de décision déterministe qui est au moins aussi bon que le randomisé en termes de nombre moyen de comparaisons.

Dans chaque nœud randomisé, prenons l'enfant ayant le plus petit sous-arbre (le sous-arbre ayant le plus petit nombre moyen de comparaisons sur un chemin menant à une feuille). Supprimons tous les autres enfants du nœud randomisé et ôtons le nœud randomisé lui-même.

L'algorithme déterministe correspondant à cet arbre modifié fonctionne encore, car l'algorithme randomisé fonctionnait quel que soit le chemin qui était emprunté à partir de chaque nœud randomisé.

Le nombre moyen de comparaisons pour l'algorithme modifié n'est pas plus grand que le nombre moyen pour l'arbre randomisé original, car nous avons supprimé les sous-arbres de moyenne supérieure dans chaque cas. En particulier, chaque fois que nous ôtons un nœud randomisé, nous laissons la moyenne générale inférieure ou égale à ce qu'elle était. En effet

- Le même ensemble de permutations qu'avant atteint le sous-arbre modifié, mais ces entrées sont traitées en un temps moyen qui est inférieur ou égal au temps d'avant, et
- Le reste de l'arbre n'est pas modifié.

L'algorithme randomisé prend donc au moins autant de temps, en moyenne, que l'algorithme déterministe correspondant. (Nous avons montré que la durée d'exécution espérée pour un tri par comparaison déterministe est $\Omega(n \lg n)$, et donc la durée espérée pour un tri par comparaison randomisé est aussi $\Omega(n \lg n)$.)

Solutions choisies du chapitre 9 : Médians et rangs

Solution de l'exercice 9.3-1

Pour les groupes de 7, l'algorithme fonctionne encore en temps linéaire. Le nombre d'éléments supérieurs à x (et, de même, le nombre d'éléments inférieurs à x) est au moins

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7}\right\rceil\right\rceil - 2\right) \geq \frac{2n}{7} - 8$$

et la récurrence devient

$T(n) \leq T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n)$,

dont on peut montrer qu'elle est $O(n)$ par substitution, comme pour les groupes de 5 dans le texte. Pour les groupes de 3, en revanche, l'algorithme ne fonctionne plus en temps linéaire. Le nombre d'éléments supérieurs à x, et le nombre d'éléments inférieurs à x, est au moins

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3}\right\rceil\right\rceil - 2\right) \geq \frac{n}{3} - 4$$

et la récurrence devient

$T(n) \leq T(\lceil n/3 \rceil) + T(2n/3 + 4) + O(n)$,

qui n'a pas de solution linéaire.

Nous pouvons démontrer que la durée dans le cas le plus défavorable pour les groupes de 3 est $\Omega(n \lg n)$. Pour ce faire, nous déduisons une récurrence pour un cas particulier qui prend un temps $\Omega(n \lg n)$.

En comptant le nombre d'éléments supérieurs à x (et, de même, le nombre d'inférieurs à x), considérons le cas particulier où il y a exactement $\left\lceil \frac{1}{2}\left\lceil \frac{n}{3}\right\rceil\right\rceil$ groupes ayant des médians $\geq x$ et où le groupe « restant » contribue pour 2 éléments supérieurs à x. Alors, le nombre d'éléments supérieurs à x est exactement

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3}\right\rceil\right\rceil - 1\right) + 1 = 2\left\lceil n/6 \right\rceil - 1$$

(le -1 décompte le groupe de x, comme d'habitude, et le $+1$ est une contribution du groupe de x), et l'étape récursive pour les éléments $\leq x$ a

$n - (2\lceil n/6 \rceil - 1) \geq n - (2(n/6 + 1) - 1) = 2n/3 - 1$ éléments.

Observons aussi que le terme $O(n)$ dans la récurrence est vraiment $\Theta(n)$, car le partitionnement de l'étape 4 prend un temps $\Theta(n)$ (et pas seulement $O(n)$). Nous obtenons donc la récurrence

$T(n) \geq T(\lceil n/3 \rceil) + T(2n/3 - 1) + \Theta(n) \geq T(n/3) + T(2n/3 - 1) + \Theta(n)$,

à partir de laquelle nous pouvons montrer que $T(n) \geq cn \lg n$ par substitution. Vous voyez aussi que $T(n)$ est non linéaire, en remarquant que chaque niveau de l'arbre récursif donne un total de n.

En fait, toute taille de groupe impaire ≥ 5 fonctionne en temps linéaire.

Solution de l'exercice 9.3-3

Une modification du tri rapide lui permettant de s'exécuter en temps $O(n \lg n)$ dans le cas le plus défavorable utilise l'algorithme PARTITION déterministe qui avait été modifié de façon à prendre comme paramètre un élément autour duquel on partitionne.

SÉLECTION prend en entrée un tableau A, les bornes p et r du sous-tableau de A et le rang i d'une statistique d'ordre ; la procédure retourne en temps linéaire par rapport à la taille du sous-tableau $A[p .. r]$ le $i$ème plus petit élément de $A[p .. r]$.

TRI-RAPIDE-CAS-OPTIMAL(A, p, r)
si $p < r$
  $i = \lfloor (r - p + 1)/2 \rfloor$
  x = SÉLECTION(A, p, r, i)
  q = PARTITION(x)
  TRI-RAPIDE-CAS-OPTIMAL($A, p, q - 1$)
  TRI-RAPIDE-CAS-OPTIMAL($A, q + 1, r$)

Pour un tableau à n éléments, le plus gros sous-tableau sur lequel TRI-RAPIDE-CAS-OPTIMAL fait de la récursivité possède $n/2$ éléments. Cette situation a lieu quand $n = r - p + 1$ est pair ; alors, le sous-tableau $A[q + 1 .. r]$ contient $n/2$ éléments et le sous-tableau $A[p .. q - 1]$ contient $n/2 - 1$ éléments.

Comme TRI-RAPIDE-CAS-OPTIMAL fait toujours de la récursivité sur des sous-tableaux dont la taille est au plus égale à la moitié de celle du tableau original, la récurrence pour la durée d'exécution dans le cas le plus défavorable est $T(n) \leq 2T(n/2) + \Theta(n) = O(n \lg n)$.

Solution de l'exercice 9.3-5

Nous supposons que nous avons une procédure MÉDIAN qui prend comme paramètres un tableau A et des indices p et r, puis retourne la valeur de l'élément médian de $A[p .. r]$ en temps $O(n)$ dans le cas le plus défavorable.

Voici un algorithme à temps linéaire SÉLECTION' qui trouve le $i$ème plus petit élément de $A[p .. r]$. Cet algorithme utilise l'algorithme PARTITION déterministe qui avait été modifié de façon à prendre comme paramètre un élément autour duquel on partitionne.

SÉLECTION'(A, p, r, i)
si $p == r$
    retourner $A[p]$
x = MÉDIAN(A, p, r)
q = PARTITION(x)
$k = q - p + 1$
si $i == k$
    retourner $A[q]$
sinon si $i < k$
    retourner SÉLECTION'(A, p, $q - 1$, i)
sinon retourner SÉLECTION'(A, $q + 1$, r, $i - k$)

Comme x est le médian de $A[p .. r]$, chacun des sous-tableaux $A[p .. q - 1]$ et $A[q + 1 .. r]$ contient au plus la moitié du nombre d'éléments de $A[p .. r]$. La récurrence pour la durée d'exécution, dans le cas le plus défavorable, de SÉLECTION' est $T(n) \leq T(n/2) + O(n) = O(n)$.

Solution du problème 9-1

Nous supposons que les nombres sont donnés dans un tableau.

a. Triez les nombres à l'aide du tri par fusion ou du tri par tas, qui prennent un temps $\Theta(n \lg n)$ dans le cas le plus défavorable. (N'employez pas le tri rapide ni le tri par insertion, qui peuvent prendre un temps $\Theta(n^2)$!) Placez les i plus grands éléments (directement accessibles dans le tableau trié) dans le tableau de sortie, ce qui prend un temps $\Theta(i)$.

Durée d'exécution totale dans le cas le plus défavorable : $\Theta(n \lg n + i) = \Theta(n \lg n)$ (car $i \leq n$).

b. Implémentez la file de priorités sous forme de tas. Construisez le tas à l'aide de Construire-TAS, ce qui prend un temps $\Theta(n)$, puis appelez EXTRAIRE-TAS-MAX i fois pour obtenir les i plus grands éléments, en temps $\Theta(i \lg n)$ dans le cas le plus défavorable, puis enfin rangez-les dans le tableau de sortie dans l'ordre inverse de leur extraction. La durée d'extraction dans le cas le plus défavorable est $\Theta(i \lg n)$ parce que

- i extractions depuis un tas à $O(n)$ éléments prennent un temps $i \cdot O(\lg n) = O(i \lg n)$, et
- la moitié des i extractions proviennent d'un tas ayant $\geq n/2$ éléments, de sorte que ces $i/2$ extractions prennent un temps $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ dans le cas le plus défavorable.

Durée d'exécution totale dans le cas le plus défavorable : $\Theta(n + i \lg n)$.

c. Utilisez l'algorithme SÉLECTION de la section 9.3 pour trouver le $i$ème plus grand nombre en temps $\Theta(n)$. Partitionnez autour de ce nombre en temps $\Theta(n)$. Triez les i plus grands nombres en temps $\Theta(i \lg i)$ dans le cas le plus défavorable (via tri par fusion ou tri par tas).

Durée d'exécution totale dans le cas le plus défavorable : $\Theta(n + i \lg i)$.

Notez que la méthode (c) est toujours asymptotiquement au moins aussi bonne que les deux autres méthodes et que la méthode (b) est asymptotiquement au moins aussi bonne que (a). (Comparer (c) et (b) est facile, mais il est moins évident de comparer (c) et (b) à (a). (c) et (b) sont asymptotiquement au moins aussi bonnes que (a), parce que n, $i \lg i$ et $i \lg n$ sont tous $O(n \lg n)$. La somme de deux trucs qui sont $O(n \lg n)$ est aussi $O(n \lg n)$.)

Solutions choisies du chapitre 11 : Tables de hachage

Solution de l'exercice 11.2-1

Pour chaque paire de clés k, l avec $k \neq l$, définissons la variable aléatoire indicatrice $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Comme nous supposons un hachage uniforme simple, $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$ et donc $\mathrm{E}[X_{kl}] = 1/m$.

Définissons maintenant la variable aléatoire Y comme étant le nombre total de collisions, de sorte que $Y = \Sigma_{k \neq l} X_{kl}$. Le nombre espéré de collisions est

$$\begin{aligned} E[Y] &= E\left[\sum_{k \neq l} X_{kl}\right] \\ &= \sum_{k \neq l} E[X_{kl}] \quad \text{(linéarité de l'espérance)} \\ &= \binom{n}{2}\frac{1}{m} \\ &= \frac{n(n-1)}{2} \cdot \frac{1}{m} \\ &= \frac{n(n-1)}{2m} \end{aligned}$$

Solution de l'exercice 11.2-4

L'indicateur dans chaque alvéole indique si elle est libre.

- Une alvéole libre est dans la liste libre, liste doublement chaînée énumérant toutes les alvéoles libres de la table. L'alvéole contient donc deux pointeurs.
- Une alvéole utilisée contient un élément et un pointeur (éventuellement NIL) vers l'élément suivant qui est haché vers cette alvéole. (Naturellement, ce pointeur pointe vers une autre alvéole de la table.)

Opérations

- ***Insertion :***
  - Si l'élément est haché vers une alvéole libre, il suffit de supprimer l'alvéole de la liste libre et d'y stocker l'élément (avec un pointeur NIL). La liste libre doit être doublement chaînée pour que cette suppression se fasse en temps $O(1)$.

- Si l'élément est haché vers une alvéole utilisée *j*, vérifiez si l'élément *x* déjà là « y est à sa place » (sa clé est également hachée vers l'alvéole *j*).
  - Si oui, ajoutez le nouvel élément à la chaîne des éléments de cette alvéole. Pour ce faire, allouez une alvéole libre (par exemple, prenez la tête de la liste libre) pour le nouvel élément et placez cette nouvelle alvéole en tête de la liste pointée par le (*j*) haché vers l'alvéole.
  - Si non, *E* appartient à la chaîne d'une autre alvéole. Déplacez-le vers une nouvelle alvéole en allouant une alvéole depuis la liste libre, en copiant le contenu (élément *x* et pointeur) de l'ancienne alvéole (de *j*) vers la nouvelle alvéole, puis en actualisant le pointeur de l'alvéole qui pointait vers *j* de façon qu'il pointe vers la nouvelle alvéole. Ensuite, insérez comme d'habitude le nouvel élément dans l'alvéole maintenant vide. Pour actualiser le pointeur pointant vers *j*, il faut le trouver en recherchant la chaîne des éléments, et ce en commençant dans l'alvéole vers laquelle est haché *x*.
- ***Suppression :*** Soit *j* l'alvéole vers laquelle est haché l'élément *x* à supprimer.
  - Si *x* est le seul élément dans *j* (*j* ne pointe pas vers d'autres rubriques), il suffit de libérer l'alvéole en la remettant en tête de la liste libre.
  - Si *x* est dans *j* mais qu'il y a un pointeur vers une chaîne d'autres éléments, déplacez vers l'alvéole *j* la première rubrique pointée et libérez l'alvéole où elle était.
  - Si vous trouvez *x* en suivant un pointeur depuis *j*, il suffit de libérer l'alvéole de *x* et de l'extirper de la chaîne (c'est-à-dire, de modifier l'alvéole qui pointait vers *x* de façon qu'elle pointe vers le successeur de *x*).
- ***Recherche :*** Vérifiez l'alvéole vers laquelle est hachée la clé et, si ce n'est pas l'élément souhaité, suivez la chaîne de pointeurs à partir de l'alvéole.

Toutes les opérations prennent des temps espérés $O(1)$ pour la même raison que dans la version présentée dans le livre : Le temps espéré pour rechercher dans les chaînes est $O(1 + \alpha)$, indépendamment de l'endroit où les chaînes sont stockées, et le fait que tous les éléments sont stockés dans la table signifie que $\alpha \leq 1$. Si la liste libre était à chaînage simple, les opérations impliquant la suppression d'une alvéole arbitraire de la liste libre ne s'exécuteraient pas en temps $O(1)$.

Solution du problème 11-2

a. Une clé particulière est hachée vers une alvéole particulière avec la probabilité $1/n$. Supposons que nous sélectionnions un ensemble particulier de k clés. La probabilité que ces k clés soient insérées dans l'alvéole en question et que toutes les autres clés soient insérées ailleurs est

$$\left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k}$$

Comme il y a $\binom{n}{k}$ façons de choisir nos k clés, nous obtenons

$$Q_k = \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} \binom{n}{k}$$

b. Pour $i = 1, 2, \ldots, n$, soit X_i une variable aléatoire indiquant le nombre de clés hachées vers l'alvéole i et soit A_i l'événement selon lequel $X_i = k$, à savoir qu'il y a exactement k clés hachées vers l'alvéole i. D'après la partie (a), nous avons $\Pr\{A\} = Q_k$. Alors,

$$\begin{aligned}
P_k &= \Pr\{M = k\} \\
&= \Pr\{(\max_{1 \le i \le n} X_i) = k\} \\
&= \Pr\{\text{il existe } i \text{ tel que } X_i = k \text{ et que } X_i \le k \text{ pour } i = 1, 2, \ldots, n\} \\
&\le \Pr\{\text{il existe } i \text{ tel que } X_i = k\} \\
&= \Pr\{A_1 \cup A_2 \cup \ldots \cup A_n\} \\
&\le \Pr\{A_1\} + \Pr\{A_2\} + \ldots + \Pr\{A_n\} \quad \text{(d'après l'inégalité (C.19))} \\
&= nQ_k .
\end{aligned}$$

c. Commençons par montrer deux faits. Primo, $1 - 1/n < 1$, ce qui implique $(1 - 1/n)^{n-k} < 1$. Secundo, $n!/(n-k)! = n \cdot (n-1) \cdot (n-2) \ldots (n-k+1) < n^k$. En combinant ces faits avec la simplification $k! > (k/e)^k$ de l'équation (3.18), nous avons

$$\begin{aligned}
Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!} \\
&< \frac{n!}{n^k k!(n-k)!} && (1 - 1/n)^{n-k} < 1 \\
&< \frac{1}{k}! && n!/(n-k)! < n^k \\
&< \frac{e^k}{k^k} && k! > (k/e)^k
\end{aligned}$$

d. Remarquez que, quand $n = 2$, $\lg \lg n = 0$, de sorte que, pour être précis, nous devons supposer que $n \ge 3$.

Dans la partie (c), nous avons montré que $Q_k < e^k/k^k$ pour tout k ; en particulier, cette inégalité est vraie pour k_0. Il suffit donc de montrer que $e^{k_0}/k_0^{k_0} < 1/n^3$ ou, ce qui revient au même, que $n^3 < k_0^{k_0}/e^{k_0}$.

En prenant les logarithmes des deux membres, nous obtenons une condition équivalente :

$$\begin{aligned}
&3 \lg n < k_0 (\lg k_0 - \lg e) \\
&= \frac{c \lg n}{\lg \lg n} (\lg c + \lg \lg n - \lg \lg \lg n - \lg e)
\end{aligned}$$

En divisant les deux membres par $\lg n$, nous obtenons la condition

$$\begin{aligned}
&3 < \frac{c}{\lg \lg n} (\lg c + \lg \lg n - \lg \lg \lg n - \lg e) \\
&= c \left(1 + \frac{\lg c - \lg e}{\lg \lg n} - \frac{\lg \lg \lg n}{\lg \lg n}\right)
\end{aligned}$$

Soit x la dernière expression entre parenthèses :

$$x = \left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right)$$

Nous devons montrer qu'il existe une constante $c > 1$ telle que $3 < cx$.

En remarquant que $\lim_{n\to\infty} x = 1$, nous voyons qu'il existe n_0 tel que $x \geq ½$ pour tout $n \geq n_0$. Par conséquent, toute constante $c > 6$ marche pour $n \geq n_0$.

Traitons les valeurs inférieures de n (en particulier, $3 \leq n < n_0$) comme suit. Comme n est obligé d'être entier, il y a un nombre fini de n dans l'intervalle $3 \leq n < n_0$. Nous pouvons évaluer l'expression x pour chacune de ces valeurs de n et déterminer une valeur de c pour laquelle $3 < cx$ pour toutes les valeurs de n. La valeur finale de c que nous utilisons est la plus grande des deux valeurs suivantes :

- 6, qui marche pour tout $n \geq n_0$, et
- $\max_{3\leq n<n_0}\{c : 3 < cx\}$, à savoir la plus grande valeur de c que nous avons choisie pour l'intervalle $3 \leq n < n_0$.

Nous avons donc montré que $Q_{k_0} < 1/n^3$, comme souhaité.

Pour voir que $P_k < 1/n^2$ pour $k \geq k_0$, observons que, d'après la partie (b), $P_k \leq nQ_k$ pour tout k. En prenant $k = k_0$, on obtient $P_{k_0} \leq nQ_{k_0} < n \cdot (1/n^3) = 1/n^2$. Pour $k > k_0$, nous montrons que nous pouvons choisir la constante c telle que $Q_k < 1/n^3$ pour tout $k \geq k_0$, d'où nous concluons que $P_k < 1/n^2$ pour tout $k \geq k_0$.

Pour choisir c comme souhaité, prenons c suffisamment grand pour que $k_0 > 3 > e$. Alors, $e/k < 1$ pour tout $k \geq k_0$, de sorte que e^k/k^k décroît quand k croît. Donc,

$$\begin{aligned} Q_k &< e^k/k^k \\ &< e^{k_0}/k^{k_0} \\ &< 1/n^3 \end{aligned}$$

pour $k \geq k_0$.

e. L'espérance de M est

$$\begin{aligned} E[M] &= \sum_{k=0}^{n} k \cdot \Pr\{M = k\} \\ &= \sum_{k=0}^{k_0} k \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} k \cdot \Pr\{M = k\} \\ &\leq \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} n \cdot \Pr\{M = k\} \\ &\leq k_0 \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + n \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\ &= k_0 \cdot \Pr\{M \leq k_0\} + n \cdot \Pr\{M > k_0\} \end{aligned}$$

ce qu'il fallait démontrer, puisque $k_0 = c \lg n / \lg\lg n$.

Pour montrer que E[M] = O(lg n / lg lg n), notons que $\Pr\{M \le k_0\} \le 1$ et que

$$\begin{aligned}
\Pr\{M > k_0\} &= \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\
&= \sum_{k=k_0+1}^{n} P_k \\
&< \sum_{k=k_0+1}^{n} 1/n^2 \qquad \text{(d'après la partie d))} \\
&< n \cdot (1/n^2) \\
&= 1/n
\end{aligned}$$

Nous en concluons que

$$\begin{aligned}
E[M] &\le k_0 \cdot 1 + n \cdot (1/n) \\
&= k_0 + 1 \\
&= \mathrm{O}(\lg n / \lg \lg n)
\end{aligned}$$

Solutions choisies du chapitre 12 : Arbres binaires de recherche

Solution de l'exercice 12.1-2

Dans un tas, la clé d'un nœud est ≥ aux deux clés de ses enfants. Dans un arbre binaire de recherche, la clé d'un nœud est ≥ à la clé de l'enfant de gauche, mais ≤ à la clé de l'enfant de droite.

La propriété de tas, contrairement à la propriété d'arbre binaire de recherche, n'aide pas à afficher les nœuds dans l'ordre trié, car elle ne dit pas quel est le sous-arbre d'un nœud qui contient l'élément à afficher avant ce nœud. Dans un tas, le plus grand élément inférieur au nœud pourrait se trouver dans n'importe lequel des sous-arbres.

Notez que, si la propriété de tas pouvait servir à afficher les clés dans l'ordre trié en temps $O(n)$, nous aurions un algorithme à temps $O(n)$ pour le tri, car la construction du tas ne prend qu'un temps $O(n)$. Mais nous savons (chapitre 8) qu'un tri par comparaison prend forcément un temps $\Omega(n \lg n)$.

Solution de l'exercice 12.2-7

Notez qu'un appel à ARBRE-MINIMUM suivi de $n - 1$ appels à ARBRE-Successeur effectue exactement le même parcours infixe de l'arbre que la procédure PARCOURS-INFIXE. PARCOURS-INFIXE affiche en premier le ARBRE-MINIMUM et, par définition, le ARBRE-Successeur d'un nœud est le nœud suivant dans l'ordre trié tel que déterminé par un parcours infixe de l'arbre.

Cet algorithme s'exécute en temps $\Theta(n)$, car :

- Il exige un temps $\Omega(n)$ pour effectuer les n appels de procédure.
- Il traverse chacun des $n - 1$ arcs de l'arbre au moins deux fois, ce qui prend un temps $O(n)$.

Pour voir que chaque arc est traversé au moins deux fois (une fois en descendant l'arbre et une fois en remontant), considérons l'arc situé entre un nœud u et l'un de ses enfants, le nœud v. En partant de la racine, on doit traverser (u, v) en descendant de u vers v, avant de le traverser en remontant de v vers u. La seule fois que l'arbre est traversé dans le sens descendant se trouve dans le code de ARBRE-MINIMUM et la seule fois que l'arbre est traversé dans le sens ascendant se trouve dans le code de ARBRE-Successeur, quand on recherche le successeur d'un nœud n'ayant pas de sous-arbre droit.

Supposons que v soit l'enfant gauche de u.

- Avant d'afficher u, nous devons afficher tous les nœuds de son sous-arbre gauche, enraciné en v, ce qui garantit la traversée descendante de l'arc (u, v).

- Après affichage de tous les nœuds du sous-arbre gauche de *u*, il faut ensuite afficher *u*. La procédure ARBRE-Successeur traverse un chemin ascendant vers *u* depuis l'élément maximum (qui n'a pas de sous-arbre droit) du sous-arbre enraciné en *v*. Ce chemin contient évidemment l'arc (*u*, *v*) et, comme tous les nœuds du sous-arbre gauche de *u* sont affichés, l'arc (*u*, *v*) n'est jamais traversé à nouveau.

Supposons maintenant que *v* soit l'enfant droit de *u*.

- Après que *u* est affiché, ARBRE-Successeur(*u*) est appelée. Pour obtenir l'élément minimum du sous-arbre droit de *u* (dont la racine est *v*), il faut traverser l'arc (*u*, *v*) vers le bas.

- Après que toutes les valeurs du sous-arbre droit de *u* ont été affichées, ARBRE-Successeur est appelée sur l'élément maximum (qui, ici aussi, n'a pas de sous-arbre droit) du sous-arbre enraciné en *v*. ARBRE-Successeur traverse un chemin ascendant vers un élément situé après *u*, puisque *u* a déjà été affiché. L'arc (*u*, *v*) doit être traversé de façon ascendante sur ce chemin et, comme tous les nœuds du sous-arbre droit de *u* ont été affichés, l'arc (*u*, *v*) n'est jamais traversé à nouveau.

Par conséquent, aucun arc n'est traversé deux fois dans le même sens.

L'algorithme s'exécute donc en temps $\Theta(n)$.

Solution de l'exercice 12.3-3

Voici l'algorithme :

```
ARBRE-TRIER(A)
soit T un arbre binaire de recherche vide
pour i = 1 à n
    ARBRE-INSÉRER(T, A[i])
PARCOURS-INFIXE(T.racine)
```

Cas le plus défavorable : $\Theta(n^2)$ — se produit quand une chaîne linéaire de nœuds résulte des opérations ARBRE-INSÉRER répétées.
Cas le plus favorable : $\Theta(n \lg n)$ — se produit quand un arbre binaire de hauteur $\Theta(\lg n)$ résulte des opérations ARBRE-INSÉRER répétées.

Solution du problème 12-2

Pour trier les chaînes de *S*, nous commençons par les insérer dans un arbre à base, puis nous employons un parcours préfixe de l'arbre pour les extraire dans l'ordre trié lexicographiquement. Le parcours de l'arbre affiche des chaînes uniquement pour les nœuds qui indiquent l'existence d'une chaîne (à savoir, les nœuds en gris clair sur la figure 12.5 du texte).

Validité

L'ordre préfixe est l'ordre correct, car :

- La chaîne d'un nœud quelconque est un préfixe de toutes les chaînes de ses descendants et apparaît donc avant eux dans l'ordre trié (règle 2).

- Les descendants de gauche d'un nœud apparaissent avant ses descendants de droite, car les chaînes correspondantes sont identiques quand on remonte vers ce nœud parent, et dans la position suivante les chaînes du sous-arbre gauche ont 0 alors que les chaînes du sous-arbre droit ont 1 (règle 1).

Durée

$\Theta(n)$.

- L'insertion prend un temps $\Theta(n)$, vu que l'insertion de chaque chaîne prend un temps proportionnel à sa longueur (traversée d'un chemin dans l'arbre dont la longueur est la longueur de la chaîne) et que la somme de toutes les longueurs de chaîne est n.
- Le parcours préfixe prend un temps $O(n)$. Il fonctionne exactement comme PARCOURS-INFIXE (il affiche le nœud courant et s'appelle de manière récursive sur les sous-arbres gauche et droit), de sorte qu'il prend un temps proportionnel au nombre de nœuds de l'arbre. Le nombre de nœuds est au plus égal à 1 plus la somme (n) des longueurs des chaînes binaires de l'arbre, car une chaîne de longueur i correspond à un chemin traversant la racine et i autres nœuds, alors qu'un nœud individuel peut être partagé entre moult chemins de chaîne.

Solutions choisies du chapitre 13 : Arbres rouge-noir

Solution de l'exercice 13.1-4

Après absorption de chaque nœud rouge dans son parent noir, le degré de chaque nœud noir est

- 2, si les deux enfants étaient déjà noirs,
- 3, si l'un des enfants était noir et l'autre rouge, ou
- 4, si les deux enfants étaient rouges.

Toutes les feuilles de l'arbre résultant ont la même profondeur.

Solution de l'exercice 13.1-5

Sur le chemin le plus long, un nœud sur deux au moins est noir. Sur le chemin le plus court, au plus chaque nœud est noir. Comme les deux chemins contiennent des nombres égaux de nœuds noirs, la longueur du plus long chemin est au plus égale à deux fois la longueur du plus court chemin.

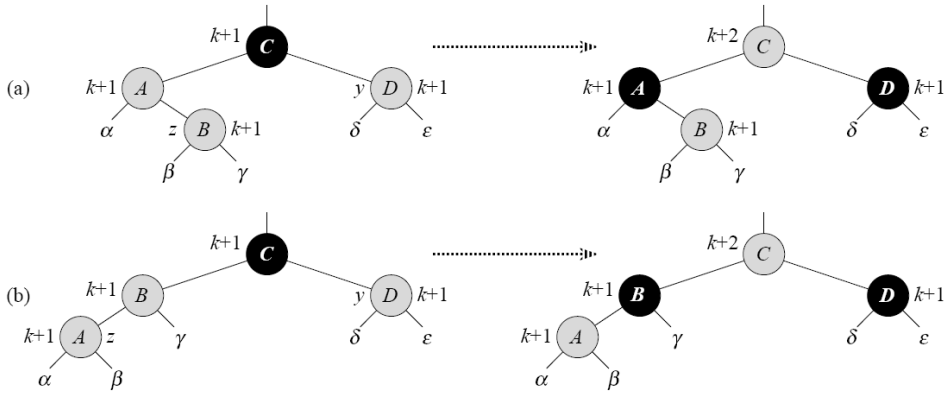
Formulé de façon plus précise, cela donne :

Comme chaque chemin contient bh(x) nœuds noirs, même le plus court chemin entre x et une feuille descendante a une longueur d'au moins bh(x). Par définition, le plus long chemin entre x et une feuille descendante a une longueur hauteur(x). Comme le plus long chemin a bh(x) nœuds noirs et qu'au moins la moitié des nœuds du plus long chemin sont noirs (d'après la propriété 4), bh(x) $\geq$ hauteur(x)/2, de sorte que

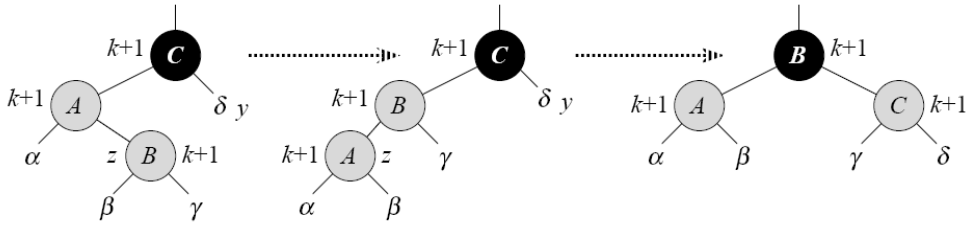
longueur du plus long chemin = hauteur(x) $\leq 2 \cdot$ bh(x) $\leq$ deux fois la longueur du plus court chemin.

Solution de l'exercice 13.3-3

Sur la figure 13.5, les nœuds A, B et D ont une hauteur noire $k + 1$ dans tous les cas, car chacun de leurs sous-arbres a une hauteur noire k et une racine noire. Le nœud C a une hauteur noire $k + 1$ sur la gauche (car ses enfants rouges ont une hauteur noire $k + 1$) et une hauteur noire $k + 2$ sur la droite (car ses enfants noirs ont une hauteur noire $k + 1$).



Sur la figure 13.6, les nœuds *A*, *B* et *C* ont une hauteur noire $k + 1$ dans tous les cas. À gauche et au milieu, chacun des sous-arbres de A et de B a une hauteur noire k et une racine noire, alors que *C* a un seul sous-arbre de ce genre et un enfant rouge de hauteur noire $k + 1$. À droite, chacun des sous-arbres de *A* et de *C* a une hauteur noire k et une racine noire, alors que les enfants rouges de *B* ont chacun une hauteur noire $k + 1$.



Cas 2 Cas 3

Les transformations conservent la propriété 5. Nous avons montré précédemment que la hauteur noire est bien définie dans les sous-arbres représentés sur les figures, de sorte que la propriété 5 est préservée dans ces sous-arbres. Il y a conservation de la propriété 5 pour l'arbre contenant les sous-arbres représentés, car chaque chemin menant dans ces sous-arbres à une feuille contribue pour $k + 2$ nœuds noirs.

Solution du problème 13-1

a. Quand on insère la clé *k*, tous les nœuds du chemin entre la racine et le nœud ajouté (une nouvelle feuille) doivent être modifiés, car le besoin d'un nouveau pointeur d'enfant se propage depuis le nouveau nœud vers tous ses ancêtres.

Quand on supprime un nœud, soit *y* le nœud réellement supprimé et soit *z* le nœud passé à la procédure de suppression.

- Si *z* a au plus un enfant, il sera extirpé, de sorte que tous les ancêtres de *z* seront modifiés. (Comme pour l'insertion, le besoin d'un nouveau pointeur d'enfant se propage vers le haut depuis le nœud supprimé.)
- Si *z* a deux enfants, alors son successeur *y* sera extirpé et déplacé vers l'emplacement de *z*. Par conséquent, tous les ancêtres de *z* et de *y* doivent être modifiés. Comme *z* est un ancêtre de *y*, on peut se contenter de dire que tous les ancêtres de *y* doivent être modifiés.

Dans l'un ou l'autre cas, les enfants de *y* (s'il y en a) ne sont pas modifiés, car nous avons supposé qu'il n'y a pas d'attribut parent.

b. Nous supposons que nous pouvons appeler deux procédures :

- CRÉER-NOUVEAU-NŒUD(*k*) crée un nœud dont l'attribut *clé* a la valeur *k* et les attributs *gauche* et *droit* la valeur NIL, puis retourne un pointeur vers le nouveau nœud.
- COPIER-NŒUD(*x*) crée un nœud dont les attributs *clé*, *gauche* et *droit* ont les mêmes valeurs que ceux du nœud x, puis retourne un pointeur vers le nouveau nœud.

Voici deux façons d'écrire ARBRE-PERSISTANT-INSÉRER. La première est une version de ARBRE-INSÉRER, modifiée de façon à créer de nouveaux nœuds le long du chemin menant vers l'emplacement du nouveau nœud, au lieu d'employer des attributs parent. Elle retourne la racine du nouvel arbre.

```
ARBRE-PERSISTANT-INSÉRER(T, k)
z = CRÉER-NOUVEAU-NŒUD(k)
nouv-racine = COPIER-NŒUD(T.racine)
y= NIL
x = nouv-racine
tant que x ≠ NIL
    y = x
    si z.clé < x.clé
            x = COPIER-NŒUD(x.gauche)
            y.gauche = x
    sinon x = COPIER-NŒUD(x.droit)
            y.droit = x
si y == NIL
    nouv-racine = z
sinon si z.clé < y.clé
    y.gauche = z
sinon y.droit = z
retourner nouv-racine
```

La seconde est une procédure récursive plutôt élégante. L'appel initial doit avoir *T.racine* comme premier argument. Elle retourne la racine du nouvel arbre.

```
ARBRE-PERSISTANT-INSÉRER(r, k)
si r == NIL
    x = CRÉER-NOUVEAU-NŒUD(k)
sinon x = COPIER-NŒUD(r)
    si k < r.clé
            x.gauche = ARBRE-PERSISTANT-INSÉRER(r.gauche, k)
    sinon x.droit = ARBRE-PERSISTANT-INSÉRER(r.droit, k)
retourner x
```

c. Comme ARBRE-INSÉRER, ARBRE-PERSISTANT-INSÉRER effectue un volume de travail constant sur chaque nœud du chemin entre la racine et le nouveau nœud. Comme la longueur du chemin est au plus *h*, elle prend un temps $O(h)$.

Comme elle alloue un nouveau nœud (volume de travail constant) pour chaque ancêtre du nœud inséré, elle exige un espace $O(h)$.

d. S'il y avait des attributs parent, alors, à cause de la nouvelle racine, il faudrait copier chaque nœud de l'arbre lors de l'insertion d'un nouveau nœud. Pour voir pourquoi, observez que les enfants de la racine seraient modifiés de façon à pointer vers la nouvelle racine, puis que leurs enfants seraient modifiés de façon à pointer vers eux, etc. Comme il y a n nœuds, avec cette version l'insertion créerait $\Omega(n)$ nouveaux nœuds et prendrait un temps $\Omega(n)$.

e. Les parties (a) et (c) ont montré que l'insertion dans un arbre binaire de recherche persistant de hauteur h, tout comme l'insertion dans un arbre binaire de recherche ordinaire, prend un temps $O(h)$ dans le cas le plus défavorable. Un arbre RN a $h = O(\lg n)$, de sorte que l'insertion dans un arbre RN ordinaire prend un temps $O(\lg n)$. Nous devons montrer que, si l'arbre RN est persistant, l'insertion peut encore se faire en un temps $O(\lg n)$. Pour ce faire, nous devons montrer deux choses :

- Comment trouver encore les pointeurs de parent dont nous avons besoin en temps $O(1)$ sans employer d'attribut parent. Nous ne pouvons pas utiliser d'attribut parent, car un arbre persistant avec attributs parent prend un temps $\Omega(n)$ pour l'insertion (voir partie (d)).
- Les modifications de nœud supplémentaires effectuées lors des opérations d'arbre RN (par rotation et recoloriage) n'entraînent pas la modification de plus de $O(\lg n)$ nœuds supplémentaires.

Voici comment trouver en temps $O(1)$ chaque pointeur de parent requis lors de l'insertion, sans avoir d'attribut parent :
Pour insérer dans un arbre RN, nous appelons RN-INSÉRER, laquelle appelle RN-INSÉRER-CORRECTION. On effectue dans RN-INSÉRER les mêmes changements que dans ARBRE-INSÉRER pour la persistance. De plus, quand RN-INSÉRER descend l'arbre pour trouver l'endroit où insérer le nouveau nœud, on lui fait construire une pile des nœuds qu'elle traverse et on passe cette pile à RN-INSÉRER-CORRECTION. RN-INSÉRER-CORRECTION a besoin de pointeurs de parent pour remonter le même chemin et, à un instant donné, elle a besoin de pointeurs de parent uniquement pour trouver le parent et le grand-parent du nœud en cours de traitement. Quand RN-INSÉRER-CORRECTION remonte la pile des parents, elle n'a besoin que de pointeurs de parent qui sont à des emplacements connus situés à une distance constante dans la pile. Donc, les données de parent peuvent être trouvées en temps $O(1)$, tout comme si nous avions stocké un attribut parent.
Voici comment la rotation et le recoloriage modifient les nœuds :

- RN-INSÉRER-CORRECTION effectue au plus 2 rotations, chacune modifiant les pointeurs d'enfant dans 3 nœuds (le nœud autour duquel on fait la rotation, le parent de ce nœud et l'un des enfants du nœud autour duquel on fait la rotation). Donc, 6 nœuds au plus sont modifiés directement par la rotation pendant RN-INSÉRER-CORRECTION. Dans un arbre persistant, tous les ancêtres d'un nœud modifié sont copiés, de sorte que les rotations de RN-INSÉRER-CORRECTION prennent un temps $O(\lg n)$ pour modifier les nœuds pour cause de rotation. (En fait, les nœuds modifiés dans ce cas se partagent un même chemin d'ancêtres de longueur $O(\lg n)$.)
- RN-INSÉRER-CORRECTION recolorie certains des ancêtres du nœud inséré, qui sont modifiés de toute façon dans l'insertion persistante, et certains enfants des ancêtres (les « oncles » dont il est question dans la description de l'algorithme). Il y a au plus $O(\lg n)$ ancêtres, donc au plus $O(\lg n)$ recoloriages d'oncle. Le recoloriage d'oncles n'entraîne pas de modifications de nœud supplémentaires pour cause de persistance, car les ancêtres des oncles sont les mêmes nœuds (ancêtres du nœud inséré) qui sont

modifiés de toute façon pour cause de persistance. Donc, le recoloriage n'affecte pas la durée d'exécution $O(\lg n)$, même avec la persistance.

Nous pourrions montrer de même que la suppression dans un arbre persistant prend aussi un temps $O(h)$ dans le cas le plus défavorable.

- Nous avons déjà vu dans la partie (a) qu'il y a $O(h)$ nœuds modifiés.
- Nous pourrions écrire une procédure RN-SUPPRIMER persistante exécutée en temps $O(h)$, en procédant aux mêmes changements que nous avons effectués pour la persistance dans l'insertion. Mais, pour ce faire sans employer de pointeurs de parent, nous devons descendre l'arbre jusqu'au nœud à supprimer, afin de construire une pile de parents comme expliqué pour l'insertion. C'est un peu rusé, si les clés de l'ensemble ne sont pas distinctes ; en effet, pour trouver le chemin vers le nœud à supprimer (un nœud particulier ayant une clé donnée), nous devons procéder à quelques changements concernant la façon dont nous stockons les choses dans l'arbre, afin de pouvoir distinguer entre les clés en double. Le plus simple est de faire en sorte que chaque clé contienne une seconde partie qui soit unique et qui permette donc de comparer les clés sans ambiguïté.

Alors, le problème consistant à montrer que la suppression n'exige qu'un temps $O(\lg n)$ dans un arbre RN persistant est le même que pour l'insertion.

- Comme pour l'insertion, nous pouvons montrer que les parents requis par RN-SUPPRIMER-CORRECTION peuvent être trouvés en temps $O(1)$ (via la même technique que pour l'insertion).
- En outre, RN-SUPPRIMER-CORRECTION effectue au plus 3 rotations, qui, comme expliqué précédemment pour l'insertion, exigent un temps $O(\lg n)$ pour modifier les nœuds pour cause de persistance. Elle effectue aussi $O(\lg n)$ modifications de couleur, qui (comme pour l'insertion) ne prennent qu'un temps $O(\lg n)$ pour modifier les ancêtres pour cause de persistance, vu que le nombre de nœuds copiés est $O(\lg n)$.

Solutions choisies du chapitre 14 : Extension des structures de données

Solution de l'exercice 14.1-7

Soit $A[1 .. n]$ le tableau de n nombres distincts.

Une façon de compter les inversions est d'additionner, pour chaque élément, le nombre d'éléments plus grands qui le précèdent dans le tableau :

$$\text{Nombre d'inversions} = \sum_{j=1}^{n} |Inv\ \ j\ |$$

où $Inv(j) = \{i : i < j \text{ et } A[i] > A[j]\}$.

Notez que $Inv(j)$ est lié au rang de $A[j]$ dans le sous-tableau $A[1 .. j]$, car les éléments de $Inv(j)$ sont la raison pour laquelle $A[j]$ n'est pas placé selon son rang. Soit $r(j)$ le rang de $A[j]$ dans $A[1 .. j]$. Alors $j = r(j) + |Inv(j)|$, de sorte que nous pouvons calculer

$$|Inv(j)| = j - r(j)$$

en insérant $A[1],\ldots, A[n]$ dans un arbre de rangs et en employant DÉTERMINER-RANG pour trouver le rang de chaque $A[j]$ dans l'arbre juste après qu'il y a été inséré. (Cette valeur DÉTERMINER-RANG est $r(j)$.)

L'insertion et DÉTERMINER-RANG prenant chacune un temps $O(\lg n)$, le temps total pour n éléments est $O(n \lg n)$.

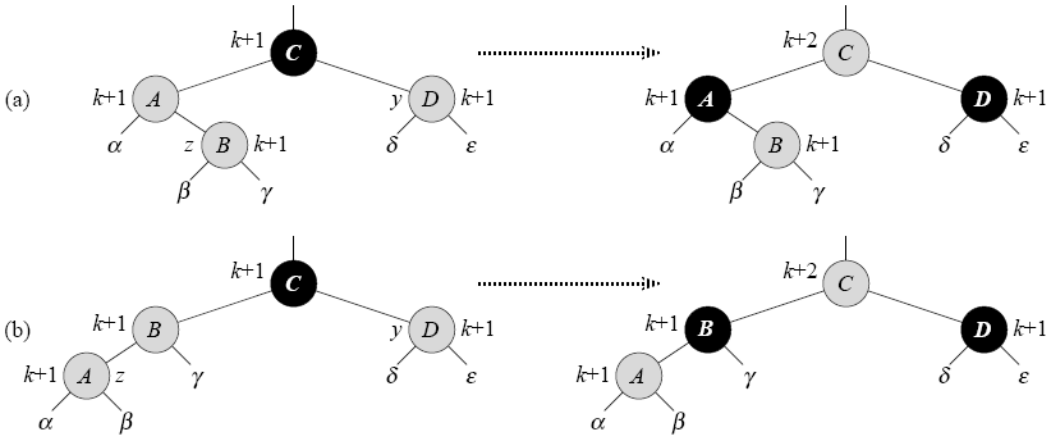
Solution de l'exercice 14.2-2

Oui, nous pouvons gérer les hauteurs noires en tant qu'attributs des nœuds d'un arbre RN sans que cela affecte les performances asymptotiques des opérations d'arbre RN. Nous faisons appel au théorème 14.1, car la hauteur noire d'un nœud peut être calculée à partir des informations du nœud et de ses deux enfants. En fait, la hauteur noire peut être calculée à partir des informations d'un seul enfant : la hauteur noire d'un nœud est la hauteur noire d'un enfant rouge, ou bien la hauteur noire d'un enfant noir plus un. Point n'est besoin de consulter le second enfant, compte tenu de la propriété 5 des arbres RN.

Les procédures RN-INSÉRER-CORRECTION et RN-SUPPRIMER-CORRECTION intègrent des recoloriages, dont chacun engendre potentiellement $O(\lg n)$ modifications de hauteur noire. Montrons que les changements de couleur dans les procédures de correction n'entraînent que des modifications de hauteur noire locales et sont donc des opérations en temps constant. Nous supposons que la hauteur noire de chaque nœud x est conservée dans l'attribut $x.bh$.

Pour RN-INSÉRER-CORRECTION, il y a 3 cas à examiner.

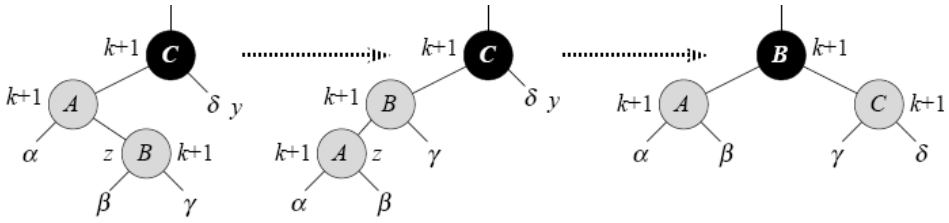
Cas 1 : l'oncle de *z* est rouge.



- Avant les modifications de couleur, supposons que tous les sous-arbres *α, β, γ, δ, ε* aient la même hauteur noire *k* avec une racine noire, de sorte que les nœuds A, B, C et D ont des hauteurs noires de *k* + 1.
- Après les modifications de couleur, le seul nœud dont la hauteur noire a changé est C. Pour corriger cela, on ajoute *z.p.p.bh* = *z.p.p.bh* + 1 après la ligne 7 dans RN-INSÉRER-CORRECTION.
- Comme le nombre de nœuds noirs entre *z.p.p* et *z* reste le même, les nœuds au-dessus de *z.p.p* ne sont pas affectés par la modification de couleur.

Cas 2 : l'oncle de *z*, *y*, est noir et *z* est un enfant de droite.

Cas 3 : l'oncle de *z*, *y*, est noir et z est un enfant de gauche.



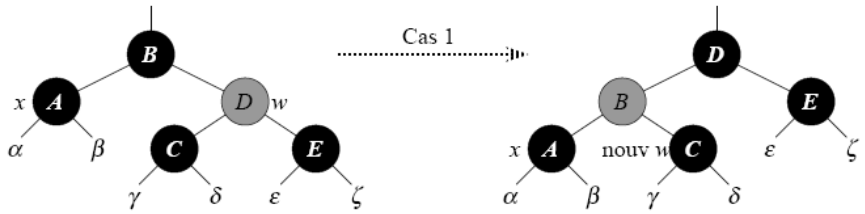
Cas 2 Cas 3

- Avec les sous-arbres *α, β, γ, δ, ε* de hauteur noire *k*, nous voyons que même avec les recoloriages et les rotations, les hauteurs noires des nœuds A, B et C restent les mêmes (*k* + 1).

Donc, RN-INSÉRER-CORRECTION conserve sa durée originale $O(\lg n)$.

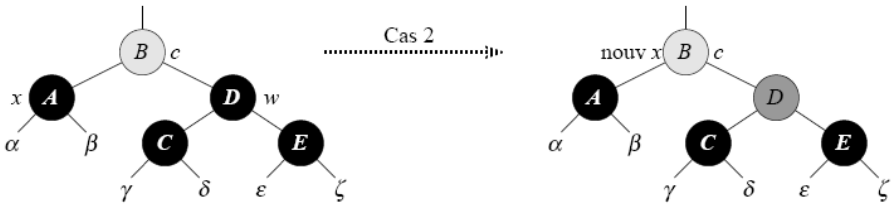
Pour RN-SUPPRIMER-CORRECTION, il y a 4 cas à examiner.

Cas 1 : le jumeau de *x*, *w*, est rouge.



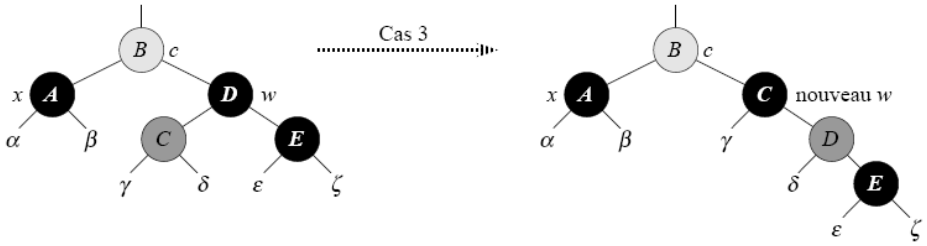
- Même si le cas 1 modifie les couleurs et effectue une rotation, les hauteurs noires ne sont pas modifiées.
- Le cas 1 modifie la structure de l'arbre, mais attend que les cas 2, 3 et 4 traitent le « noir en plus » sur x.

Cas 2 : le jumeau de *x*, *w*, est noir et les deux enfants de *w* sont noirs.



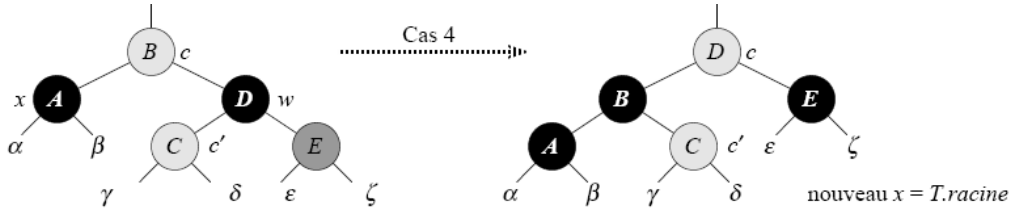
- *w* est colorié en rouge et le noir « en plus » de *x* est remonté en *x.p*.
- Nous pouvons maintenant ajouter *x.p.bh* = *x.bh* après la ligne 10 dans RN-SUPPRIMER-CORRECTION.
- C'est une modification en temps constant. Ensuite, on continue à boucler pour traiter le noir en plus sur *x.p*.

Cas 3 : le jumeau de *x*, *w*, est noir, l'enfant gauche de *w* est rouge et l'enfant droit de *w* est noir.



- Indépendamment des modifications de couleur et de la rotation de ce cas, les hauteurs noires ne changent pas.
- Comme le cas 3 se contente de configurer la structure de l'arbre, il peut sans problème continuer vers le cas 4.

Cas 4 : le jumeau de *x*, *w*, est noir et l'enfant droit de *w* est rouge.



- Les nœuds A, C et E conservent les mêmes sous-arbres, de sorte que leurs hauteurs noires ne changent pas.
- Ajoutez ces deux assignations en temps constant dans RN-SUPPRIMER-CORRECTION après la ligne 20 :

  $x.p.bh = x.bh + 1$
  $x.p.p.bh = x.p.bh + 1$
- Le noir en plus est définitivement traité. La boucle se termine.

Donc, RN-SUPPRIMER-CORRECTION conserve sa durée originale $O(\lg n)$.

Par conséquent, nous concluons que les hauteurs noires des nœuds peuvent être gérées en tant qu'attributs dans les arbres RN sans que cela affecte les performances asymptotiques des opérations d'arbre RN.

Pour ce qui concerne la seconde partie de la question, non, nous ne pouvons pas gérer les profondeurs de nœud sans que cela affecte les performances asymptotiques des opérations d'arbre RN. La profondeur d'un nœud dépend de celle de son parent. Quand la profondeur d'un nœud est modifiée, les profondeurs de tous les nœuds situés au-dessous de lui dans l'arbre doivent être mises à jour. L'actualisation du nœud racine entraîne qu'il faut actualiser $n - 1$ autres nœuds, ce qui signifierait que les opérations sur l'arbre qui modifient les profondeurs de nœud ne pourraient pas s'exécuter en temps $O(n \lg n)$.

Solution de l'exercice 14.3-7

Idée générale : Déplacez une ligne de balayage de gauche à droite, tout en gérant l'ensemble des rectangles couramment intersectés par la ligne dans un arbre d'intervalles. Cet arbre organise tous les rectangles dont l'intervalle x inclut la position courante de la ligne et s'appuie sur les intervalles y des rectangles, de sorte que tous les intervalles y de l'arbre qui se recoupent correspondent aux rectangles qui se recoupent.

Détails :

1 Triez les rectangles en fonction de leurs coordonnées x. (En fait, chaque rectangle doit apparaître deux fois dans la liste triée, une fois pour sa coordonnée x gauche et une fois pour sa coordonnée x droite.)

2 Lisez la liste triée (dans l'ordre croissant des coordonnées x).
- Quand vous trouvez une coordonnée x d'un coin gauche, vérifiez si la coordonnée y du rectangle recoupe un intervalle de l'arbre et insérez le rectangle (indexé sur son intervalle de coordonnées y) dans l'arbre.
- Quand vous trouvez une coordonnée x d'un coin droit, supprimez le rectangle de l'arbre d'intervalles.

L'arbre d'intervalles contient toujours l'ensemble des rectangles « ouverts » intersectés par la ligne de balayage. Si vous trouvez un recoupement dans l'arbre, c'est qu'il y a des rectangles qui se recoupent.

Durée : $O(n \lg n)$
- $O(n \lg n)$ pour trier les rectangles (via tri par fusion ou tri par tas).
- $O(n \lg n)$ pour les opérations d'arbre d'intervalles (insertion, suppression et recherche de recoupement).

Solutions choisies du chapitre 15 : Programmation dynamique

Solution de l'exercice 15.2-5

Pour chaque exécution de la boucle *l*, il y a $n - l + 1$ exécutions de la boucle *i*. Pour chaque exécution de la boucle *i*, il y a $j - i = l - 1$ exécutions de la boucle *k*, référençant à chaque fois *m* deux fois. Donc, le nombre total de fois qu'une entrée de *m* est référencée lors du calcul d'autres entrées est $\sum_{l=2}^{n}(n-l+1)(l-1)2$

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) &= \sum_{l=2}^{n}(n-l+1)(l-1)2 \\
&= 2\sum_{l=1}^{n-1}(n-l)l \\
&= 2\sum_{l=1}^{n-1}nl - 2\sum_{l=1}^{n-1}l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3}
\end{aligned}$$

Solution de l'exercice 15.3-1

Exécuter CHAÎNE-MATRICES-RÉCURSIF est asymptotiquement plus efficace que d'énumérer toutes les façons de parenthéser le produit et de calculer le nombre de multiplications pour chacune.

Considérons le traitement des sous-problèmes par les deux approches.

- Pour chaque emplacement possible de fractionnement de la chaîne de matrices, l'approche énumérative trouve toutes les façons de parenthéser la moitié gauche, trouve toutes les façons de parenthéser la moitié droite et recherche toutes les combinaisons possibles de la moitié gauche et de la moitié droite. Le volume de travail pour regarder chaque combinaison de résultats du sous-problème de moitié gauche et du sous-problème de moitié droite est donc le produit du nombre de façons de faire la moitié gauche et du nombre de façons de faire la moitié droite.

- Pour chaque emplacement possible de fractionnement de la chaîne de matrices, CHAÎNE-MATRICES-RÉCURSIF trouve la meilleure façon de parenthéser la moitié gauche, trouve la meilleure façon de parenthéser la moitié droite, puis se contente de combiner ces deux résultats. Par conséquent, le volume de travail pour combiner les résultats du sous-problème de moitié gauche et du sous-problème de moitié droite est O(1).

La section 15.2 a prouvé que la durée d'exécution de l'énumération est $\Omega(4^n/n^{3/2})$. Nous allons montrer que la durée d'exécution de CHAÎNE-MATRICES-RÉCURSIF est $O(n3^{n-1})$.

Pour obtenir une borne supérieure de la durée d'exécution de CHAÎNE-MATRICES-RÉCURSIF, nous utiliserons la même approche que dans la section 15.2 pour obtenir une borne inférieure : dériver une récurrence de la forme $T(n) \leq \ldots$ et la résoudre par substitution. Pour la récurrence de la borne inférieure, le livre supposait que l'exécution des lignes 1–2 et 6–7 prenait chacune un temps unitaire au moins. Pour la récurrence de la borne supérieure, nous supposerons que ces paires de lignes prennent chacune au plus un temps constant c. Nous avons donc la récurrence

$$T(n) \leq \begin{cases} c & \text{si } n = 1 \\ c + \sum_{k=1}^{n-1} (T(k) + T(n-k) + c) & \text{si } n \geq 2 \end{cases}$$

C'est exactement la récurrence donnée dans le livre, sauf qu'il y a c à la place de 1 ; nous pouvons donc la réécrire sous la forme

$$T(n) \leq 2\sum_{i=1}^{n-1} T(i) + cn$$

Nous démontrerons que $T(n) = O(n3^{n-1})$ en employant la méthode de substitution. (Remarque : N'importe quelle borne supérieure de $T(n)$ qui est $o(4^n/n^{3/2})$ suffira. Vous préférerez peut-être en prouver une qui soit plus simple à imaginer, par exemple $T(n) = O(3.5^n)$.) Plus précisément, nous allons montrer que $T(n) \leq cn3^{n-1}$ pour tout $n \geq 1$. Le cas de base est facile, car $T(1) \leq c = c \cdot 1 \cdot 3^{1-1}$. Pour la récurrence, pour $n \geq 2$ nous avons

$$\begin{aligned}
T(n) &\leq 2\sum_{i=1}^{n-1} T(i) + cn \\
&\leq 2\sum_{i=1}^{n-1} ci3^{i-1} + cn \\
&\leq c \cdot \left(2\sum_{i=1}^{n-1} i3^{i-1} + n \right) \\
&= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{3-1^{\,2}} \right) + n \right) \qquad \text{(voir ci-après)} \\
&= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n \right) \\
&= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\
&\leq cn3^{n-1} \text{ pout tout } c > 0,\ n \geq 1
\end{aligned}$$

L'exécution de CHAÎNE-MATRICES-RÉCURSIF prend un temps $O(n3^{n-1})$ et l'énumération de tous les parenthésages prend un temps $\Omega(4^n/n^{3/2})$, de sorte que CHAÎNE-MATRICES-RÉCURSIF est plus efficace.

Remarque : La substitution donnée en amont utilise le fait suivant :

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}$$

Cette équation découle de l'équation (A.5) via dérivation. Soit

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1$$

Alors

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}$$

Solution de l'exercice 15.4-4

Quand on calcule un enregistrement particulier de la table c, on n'a pas besoin des enregistrements situés avant l'enregistrement précédent. À un instant donné, on n'a donc besoin de garder en mémoire que deux enregistrements ($2 \cdot Y.longueur$ éléments). (Remarque : Chaque enregistrement de c possède en fait $Y.longueur + 1$ éléments, mais point n'est besoin de stocker la colonne de 0 ; à la place, on peut faire « savoir » au programme que ces éléments sont 0.) Sachant cela, nous n'avons besoin que de $2 \cdot \min(m, n)$ éléments, si nous appelons toujours LONGUEUR-PLSC avec la séquence la plus courte comme argument Y.

Nous pouvons donc traiter la table c de la manière suivante :

- Utiliser deux tableaux de longueur $\min(m, n)$, *enreg-prec* et *enreg-courant*, pour contenir les enregistrements idoines de c.
- Initialiser *enreg-prec* à 0 partout et calculer *enreg-courant* de la gauche vers la droite.
- Quand *enreg-courant* est rempli, s'il reste d'autres enregistrements à calculer, copier *enreg-courant* dans *enreg-prec* et calculer le nouvel new *enreg-courant*.

En fait, pendant le calcul on n'a besoin que d'un petit peu plus des éléments d'un seul enregistrement ($\min(m, n) + 1$ éléments) de c. Les seuls éléments requis dans la table quand il est temps de calculer $c[i, j]$ sont $c[i, k]$ pour $k \leq j - 1$ (c'est-à-dire, les éléments antérieurs de l'enregistrement courant dont on aura besoin pour calculer l'enregistrement suivant) ; et $c[i - 1, k]$ pour $k \geq j - 1$ (c'est-à-dire, les éléments de l'enregistrement précédent dont on a encore besoin pour calculer le reste de l'enregistrement courant). C'est un élément pour chaque k de 1 à $\min(m, n)$, sauf qu'il y a deux éléments avec $k = j - 1$, d'où l'élément supplémentaire requis en plus du nombre d'éléments d'un seul enregistrement.

Nous pouvons donc traiter la table c de la manière suivante :

- Employer un tableau a de longueur $\min(m, n) + 1$ pour contenir les éléments idoines de c. Quand il faudra calculer $c[i, j]$, a contiendra les éléments suivants :
  - $a[k] = c[i, k]$ pour $1 \leq k < j - 1$ (c'est-à-dire, les éléments antérieurs de « l'enregistrement » courant),
  - $a[k] = c[i - 1, k]$ pour $k \geq j - 1$ (c'est-à-dire, les éléments de l'enregistrement précédent),

- $a[0] = c[i, j - 1]$ (c'est-à-dire, l'élément précédent calculé, qui ne pourrait pas être mis au « bon » emplacement dans a sans qu'il faille effacer $c[i - 1, j - 1]$, dont on a encore besoin).

- Initialiser a à 0 partout et calculer les éléments de la gauche vers la droite.
  - Remarquez que les 3 valeurs requises pour calculer $c[i, j]$ pour $j > 1$ sont dans $a[0] = c[i, j - 1]$, $a[j - 1] = c[i - 1, j - 1]$ et $a[j] = c[i - 1, j]$.
  - Quand $c[i, j]$ a été calculé, mettre $a[0]$ ($c[i, j - 1]$) à sa « bonne » place, $a[j - 1]$, et mettre $c[i, j]$ dans $a[0]$.

Solution du problème 15-4

Remarque : Nous supposons qu'il n'y a pas de mot qui soit plus long que ce que peut contenir une ligne, c'est-à-dire $l_i \leq M$ pour tout i.
Primo, nous avons besoin de certaines définitions pour énoncer le problème de manière plus uniforme. Ces définitions intégreront les cas spéciaux concernant la dernière ligne et le fait de savoir si une séquence de mots tient sur une ligne, de sorte que nous pourrons les ignorer quand nous élaborerons notre stratégie générale.

- Soit $extras[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ le nombre d'espaces supplémentaires à la fin d'une ligne contenant les mots i à j. Notez que *extras* peut être négatif.
- Définissons maintenant le coût d'inclusion d'une ligne contenant les mots i à j dans la somme à minimiser :

$$lc[i,j] = \begin{cases} \infty & \text{si } extras[i,j] < 0 \quad \text{(c'est-à-dire, si mots } i,\ldots, j \text{ ne tiennent pas)} ; \\ 0 & \text{si } j = n \text{ et } extras[i,j] \geq 0 \quad \text{(la dernière ligne coute 0)} ; \\ (extras[i,j])^3 & \text{sinon.} \end{cases}$$

  En rendant infini le coût d'une ligne quand les mots ne tiennent pas dedans, nous empêchons une telle configuration de faire partie d'une somme minimale et, en rendant nul le coût de la dernière ligne (si les mots y tiennent), nous empêchons la disposition de la dernière ligne d'influencer la somme à minimiser.

Nous voulons minimiser la somme de *lc* sur l'ensemble des lignes du paragraphe.
Nos sous-problèmes sont : comment optimiser la disposition des mots 1,…, j avec $j = 1,\ldots, n$.
Considérons une disposition optimale des mots 1,…, j. Supposons que nous sachions que la dernière ligne, qui se termine au mot j, commence par le mot i. Les lignes précédentes contiennent donc les mots 1,…, i - 1. En fait, elles doivent contenir une disposition optimale des mots 1,…, $i - 1$. (La démonstration usuelle de type couper-coller s'applique ici.)
Soit $c[j]$ le coût d'une disposition optimale des mots 1,…, j. Si nous savons que la dernière ligne contient les mots i,…, j, alors $c[j] = c[i - 1] + lc[i, j]$. Comme cas de base, quand nous calculons $c[1]$, nous avons besoin de $c[0]$. Si nous faisons $c[0] = 0$, alors $c[1] = lc[1, 1]$, ce qui est bien ce que nous désirons.

Mais, naturellement, nous devons trouver quel est le mot qui commence la dernière ligne pour le sous-problème des mots 1,…, j. Nous essayons donc toutes les possibilités pour le mot i et prenons celle qui donne le moindre coût. Ici, i va de 1 à j.

Nous pouvons donc définir $c[j]$ récursivement par

$$c[j] = \begin{cases} 0 & \text{si } j = 0 \\ \min\limits_{1 \le i \le j} \{c[i-1] + lc[i,j]\} & \text{si } j > 0 \end{cases}$$

Remarquez que la façon dont nous avons défini *lc* garantit que

- Tous les choix effectués tiendront sur la ligne (car une disposition avec $lc = \infty$ ne peut pas être choisie comme minimum), et
- Le coût de placement des mots $i,\ldots, j$ sur la dernière ligne ne vaudra pas 0, sauf si c'est vraiment la dernière ligne du paragraphe ($j = n$) ou si les mots $i,\ldots, j$ remplissent toute la ligne.

Nous pouvons calculer une table de valeurs *c* de la gauche vers la droite, car chaque valeur dépend uniquement des valeurs précédentes.
Pour savoir quel mot va sur quelle ligne, nous pouvons gérer une table *p* parallèle qui pointe vers l'endroit d'où provenait chaque valeur *c*. Quand on calcule $c[j]$, si $c[j]$ dépend de la valeur de $c[k-1]$, on fait $p[j] = k$. Ensuite, une fois calculé $c[n]$, on peut suivre les pointeurs pour voir où placer les sauts de ligne. La dernière ligne commence au mot $p[n]$ et va jusqu'au mot *n*. La ligne précédente commence au mot $p[p[n]]$ et va jusqu'au mot $p[n] - 1$, etc.
Dans le pseudo-code, voici comment nous construisons les tables :

```
AFFICHER-PROPREMENT(l, n, M)
soient extras[1 .. n, 1 .. n], lc[1 .. n, 1 .. n] et c[0 .. n] de nouveaux tableaux
// Calculer extras[i, j] pour 1 ≤ i ≤ j ≤ n.
pour i = 1 à n
    extras[i, i] = M – l_i
    pour j = i + 1 à n
            extras[i, j] = extras[i, j - 1] – l_j – 1
// Calculer lc[i, j] pour 1 ≤ i ≤ j ≤ n.
pour i = 1 à n
    pour j = i à n
            si extras[i, j] < 0
                    lc[i, j] = ∞
            sinon si j == n et extras[i, j] ≥ 0
                    lc[i, j] = 0
            sinon lc[i, j] = (extras[i, j])^3
// Calculer c[j] et p[j] pour 1 ≤ j ≤ n.
c[0] = 0
pour j = 1 à n
    c[j] = ∞
    pour i = 1 à j
            si c[i - 1] + lc[i, j] < c[j]
                    c[j] = c[i - 1] + lc[i, j]
                    p[j] = i
retourner c et p
```

Très clairement, tant la durée que l'espace sont $\Theta(n^2)$.
En fait, nous pouvons faire un peu mieux : nous pouvons faire tomber la durée et l'espace

à $\Theta(nM)$. L'observation fondamentale est qu'au plus $\lceil M/2 \rceil$ mots tiennent sur une ligne. (Chaque mot fait au moins un caractère et il y a un espace entre deux mots.) Comme une ligne contenant les mots $i,\ldots, j$ contient $j - i + 1$ mots, si $j - i + 1 > \lceil M/2 \rceil$ nous savons que $lc[i, j] = \infty$. Nous avons seulement besoin de calculer et stocker $extras[i, j]$ et $lc[i, j]$ pour $j - i + 1 \leq \lceil M/2 \rceil$. Et l'en-tête de la boucle **pour** interne dans le calcul de $c[j]$ et de $p[j]$ peut aller de $\max(1, j - \lceil M/2 \rceil + 1)$ à j.

Nous pouvons même faire tomber l'espace à $\Theta(n)$. Pour ce faire, nous ne stockons pas les tables *lc* et *extras*, mais plutôt calculons la valeur de $lc[i, j]$ comme exigé dans la dernière boucle. L'idée est que nous pourrions calculer $lc[i, j]$ en temps $O(1)$ si nous connaissions la valeur de $extras[i, j]$. Et si nous recherchons la valeur minimum dans l'ordre *décroissant* de i, nous pouvons la calculer comme $extras[i, j] = extras[i + 1, j] - l_i - 1$. (Initialement, $extras[j, j] = M - l_j$.) Cette amélioration fait tomber l'espace à $\Theta(n)$, car maintenant les seules tables stockées sont c et p.

Voici comment indiquer que tel mot va sur telle ligne. La sortie de DONNER-LIGNES(p, j) est une séquence de triplets (k, i, j), indiquant que les mots $i,\ldots, j$ sont affichés sur la ligne k. La valeur de retour est le numéro de ligne k.

```
DONNER-LIGNES(p, j)
i = p[j]
si i == 1
    k = 1
sinon k = DONNER-LIGNES(p, i – 1)
afficher(k, i, j)
retourner k
```

L'appel initial est DONNER-LIGNES(p, n). Comme la valeur de j diminue à chaque appel récursif, DONNER-LIGNES prend un temps total $O(n)$.

Solutions choisies du chapitre 16 : Algorithmes gloutons

Solution de l'exercice 16.1-4

Soit S l'ensemble des n activités.

La solution « évidente » consistant à employer CHOIX-D'ACTIVITÉS-GLOUTON pour trouver pour la première salle un ensemble maximum S_1 d'activités compatibles prises dans S, puis à utiliser de nouveau CHOIX-D'ACTIVITÉS-GLOUTON pour trouver pour la deuxième salle un ensemble maximum S_2 d'activités compatibles prises dans $S - S_1$, etc. jusqu'à assignation de toutes les activités, exige un temps $\Theta(n^2)$ dans le cas le plus défavorable. En outre, cela peut produire un résultat utilisant plus de salles que nécessaire. Considérons les activités ayant les intervalles {[1, 4), [2, 5), [6, 7), [4, 8)}. CHOIX-D'ACTIVITÉS-GLOUTON choisirait les activités ayant les intervalles [1, 4) et [6, 7) pour la première salle, puis chacune des activités ayant les intervalles [2, 5) et [4, 8) devrait aller dans sa propre salle, ce qui ferait un total de trois salles. Une solution optimale placerait les activités ayant les intervalles [1, 4) et [4, 8) dans une salle et les activités ayant les intervalles [2, 5) et [6, 7) dans une autre salle, ce qui ferait un total de seulement deux salles.

Il existe, cependant, un algorithme correct, dont la durée asymptotique est seulement la durée requise pour trier les activités par heure : $O(n \lg n)$ pour les heures quelconques, voire $O(n)$ pour les heures qui sont de petits entiers.

L'idée générale est de parcourir les activités par ordre d'heure de début, en assignant chacune à une salle libre à cette heure. Pour ce faire, balayez l'ensemble des événements consistant en activités qui démarrent et en activités qui finissent, et ce par ordre d'heure d'événement. Gérez deux listes de salles : les salles qui sont occupées à l'heure d'événement courante t (car on leur a assigné une activité i commençant à $s_i \leq t$ et finissant à $f_i > t$) et les salles qui sont libres à l'heure t. (Comme dans le problème de choix d'activités de la section 16.1, nous supposons que les intervalles d'heures d'activité sont semi ouverts, c'est-à-dire que, si $s_i \geq f_j$, alors les activités i et j sont compatibles.) Quand t est l'heure de début d'une certaine activité, assignez cette activité à une salle libre et transférez la salle de la liste libre vers la liste occupée. Quand t est l'heure de fin d'une certaine activité, transférez la salle de la liste occupée vers la liste libre. (L'activité est certainement dans une certaine salle, car les heures d'événement sont traitées dans l'ordre et l'activité doit avoir démarré avant son heure de fin t, donc doit avoir été assignée à une salle.)

Pour ne pas utiliser plus de salles que nécessaire, sélectionnez systématiquement une salle ayant déjà servi pour une activité avant que de choisir une salle n'ayant jamais servi. (Cela peut se faire en travaillant toujours en tête de la liste des salles libres (en plaçant les salles libérées en tête de la liste et en choisissant les salles en début de liste), afin qu'une

nouvelle salle ne vienne pas en tête et soit choisie, s'il y a des salles précédemment utilisées.) Cela garantit que l'algorithme emploiera le minimum de salles. L'algorithme se terminera sur un emploi du temps exigeant $m \leq n$ salles. Soit i la première activité planifiée dans la salle m. La raison pour laquelle i a été mise dans la m-ième salle est que les $m - 1$ premières salles étaient occupées à l'heure s_i. Donc, à cette heure il y a m activités simultanées. Par conséquent, tout emploi du temps doit utiliser au moins m salles, de sorte que l'emploi du temps produit par l'algorithme est optimal.
Durée d'exécution :

- Trier les $2n$ événements début/fin d'activité. (Dans l'ordre trié, un événement fin d'activité doit précéder un événement début d'activité ayant la même heure.) Durée $O(n \lg n)$ pour les heures quelconques, voire $O(n)$ si les heures sont restreintes (par exemple, si ce sont de petits entiers).
- Traiter les événements en temps $O(n)$: balayer les $2n$ événements, en effectuant un travail $O(1)$ pour chacun (transfert d'une salle entre les deux listes et association éventuelle d'une activité à cette salle).

Total : $O(n$ + durée de tri$)$

Solution de l'exercice 16.2-2

La solution s'appuie sur l'observation de sous-structure optimale dans le texte : soit i l'article de numéro maximal dans une solution optimale S pour le poids W et les articles $1,\ldots, n$. Alors $S' = S - \{i\}$ est forcément une solution optimale pour le poids $W - w_i$ et les articles $1,\ldots, i - 1$, et la valeur de la solution S est v_i plus la valeur de la solution du sous-problème S'.

Nous pouvons exprimer cette relation dans la formule suivante : soit $c[i, w]$ la valeur de la solution pour les articles $1,\ldots, i$ et le poids maximum w. Alors

$$c[i,w] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } w = 0 \\ c[i-1,w] & \text{si } w_i > w \\ \max(v_i + c[i-1,w-w_i], c[i-1,w]) & \text{si } i > 0 \text{ et } w \geq w_i \end{cases}$$

Le dernier cas stipule que la valeur d'une solution pour i articles soit inclut l'article i, auquel cas c'est v_i plus une solution de sous-problème pour $i - 1$ articles et le poids excluant w_i, soit n'inclut pas l'article i, auquel cas c'est une solution de sous-problème pour $i - 1$ articles et le même poids. Autrement dit, si le voleur emporte l'article i, il prend la valeur v_i et il peut choisir parmi les articles $1,\ldots, i - 1$ dans la limite de poids $w - w_i$ pour obtenir la valeur supplémentaire $c[i - 1, w - w_i]$. Si, en revanche, il décide de ne pas prendre l'article i, il peut choisir parmi les articles $1,\ldots, i - 1$ dans la limite de poids w et obtenir la valeur $c[i - 1, w]$. Il faut effectuer le meilleur de ces deux choix.

L'algorithme prend en entrée le poids maximum W, le nombre d'articles n et les deux séquences $v = \langle v_1, v_2,\ldots, v_n\rangle$ et $w = \langle w_1, w_2,\ldots, w_n\rangle$. Il stocke les valeurs $c[i, j]$ dans une table $c[0 .. n, 0 .. W]$, dont les éléments sont calculés par ordre majeur de ligne. (Autrement dit, la première ligne de c est remplie de la gauche vers la droite, puis la deuxième ligne, etc.) À la fin du calcul, $c[n, W]$ contient la valeur maximum que le voleur peut prendre.

SAC-01-DYNAMIQUE(v, w, n, W)
soit $c[0 .. n, 0 .. W]$ un nouveau tableau
pour $w = 0$ **à** W
    $c[0, w] = 0$
pour $i = 1$ **à** n
    $c[i, 0] = 0$
    pour $w = 1$ **à** W
        si $w_i \leq w$
            si $v_i + c[i - 1, w - w_i] > c[i - 1, w]$
                $c[i, w] = v_i + c[i - 1, w - w_i]$
            sinon $c[i, w] = c[i - 1, w]$
        sinon $c[i, w] = c[i - 1, w]$

Nous pouvons employer la table c pour calculer l'ensemble des articles en commençant à $c[n, W]$ et en remontant jusqu'aux origines des valeurs optimales. Si $c[i, w] = c[i - 1, w]$, alors l'article i n'appartient pas à la solution et nous continuons à remonter avec $c[i - 1, w]$. Sinon, l'article i appartient à la solution et nous continuons à remonter avec $c[i - 1, w - w_i]$.

L'algorithme précédent prend un temps total $\Theta(nW)$:

- $\Theta(nW)$ pour remplir la table c : $(n + 1) \cdot (W + 1)$ éléments, chacun exigeant un temps de calcul $\Theta(1)$.
- $O(n)$ pour tracer la solution (vu qu'elle commence à la ligne n de la table et qu'elle remonte d'une ligne à chaque étape).

Solution de l'exercice 16.2-7

Triez A et B par ordre monotone décroissant.

Voici une démonstration de ce que cette méthode donne une solution optimale. Considérons des indices i et j tels que $i < j$, ainsi que les termes $a_i^{b_i}$ et $a_j^{b_j}$. Nous voulons montrer qu'il n'est pas pire d'inclure ces termes dans l'indemnité que d'inclure $a_i^{b_j}$ et $a_j^{b_i}$, c'est-à-dire que $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$. Comme A et B sont triés par ordre monotone décroissant et que $i < j$, nous avons $a_i \geq a_j$ et $b_i \geq b_j$. Comme a_i et a_j sont positifs et que $b_i - b_j$ n'est pas négatif, nous avons $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. En multipliant les deux côtés par $a_i^{b_j} a_j^{b_j}$, nous obtenons $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

Comme l'ordre de la multiplication n'a pas d'importance, le tri de A et B par ordre monotone croissant fonctionne également.

Solutions choisies du chapitre 17 : Analyse amortie

Solution de l'exercice 17.1-3

Soit c_i le coût de la $i$ème opération.

$$c_i = \begin{cases} i & \text{si } i \text{ est une puissance exacte de 2,} \\ 1 & \text{sinon.} \end{cases}$$

Opération	Coût
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
.	.
.	.
.	.

n opérations coûtent

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n-1) < 3n$$

(Remarque : Nous ignorons la partie entière inférieure dans la borne supérieure de $\sum 2^j$.)

Coût moyen d'opération = Coût total / Nb. opérations < 3.
D'après l'analyse par agrégats, le coût amorti par opération est $O(1)$.

Solution de l'exercice 17.2-2

Soit c_i le coût de la $i$ème opération.

$$c_i = \begin{cases} i & \text{si } i \text{ est une puissance exacte de 2,} \\ 1 & \text{sinon.} \end{cases}$$

On facture chaque opération 3 euros (coût amorti $\hat{c}_i$).

- Si i n'est pas une puissance exacte de 2, on paie 1 euro et on enregistre 2 euros comme crédit.
- Si i est une puissance exacte de 2, on paie i euros à l'aide du crédit enregistré.

Opération	Coût	Coût réel	Crédit restant
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
.	.	.	.
.	.	.	.
.	.	.	.

Comme le coût amorti est 3 euros par opération,

$$\sum_{i=1}^{n} \hat{c}_i = 3n$$

L'exercice 17.1-3 a montré que

$$\sum_{i=1}^{n} c_i < 3n$$

Nous avons alors

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

$\Rightarrow$ crédit = coût amorti – coût réel ≥ 0.

Comme le coût amorti de chaque opération est $O(1)$ et que la quantité de crédit ne devient jamais négative, le coût total de n opérations est $O(n)$.

Solution de l'exercice 17.2-3

Nous introduisons un nouveau champ *A.max* pour contenir l'indice du 1 de poids fort dans *A*. Initialement *A.max* est réglé sur –1, car le bit de poids faible de *A* est à l'indice 0 et il n'y a pas initialement de 1 dans *A*. La valeur de *A.max* est mise à jour de façon idoine, quand le compteur est incrémenté ou réinitialisé, et nous utilisons cette valeur pour limiter la portion de *A* à examiner pour le réinitialiser. En contrôlant ainsi le coût de RÉINITIALISER, nous pouvons le restreindre à un montant susceptible d'être couvert par le crédit provenant des INCRÉMENTER antérieurs.

```
INCRÉMENTER(A)
i = 0
tant que i < A.longueur et A[i] == 1
```

```
        A[i] = 0
        i = i + 1
    si i < A.longueur
        A[i] = 1
        // Début des ajouts à la procédure INCRÉMENTER du livre.
        si i > A.max
                A.max = i
    sinon A.max = –1

RÉINITIALISER(A)
pour i = 0 à A.max
    A[i] = 0
A.max = –1
```

Comme pour le compteur du livre, nous supposons que cela coûte 1 euro de basculer un bit. En outre, nous supposons que cela coûte 1 euro de mettre à jour *A.max*.

L'initialisation et la réinitialisation des bits par INCRÉMENTER fonctionne exactement comme pour le compteur original du livre : 1 euro paie l'initialisation d'un bit à 1 ; 1 euro sera placé sur le bit qui est initialisé à 1 comme crédit ; le crédit placé sur chaque bit à 1 paiera la réinitialisation du bit lors de l'incrémentation.

En outre, nous utiliserons 1 euro pour payer la mise à jour de *max* et, si *max* augmente, nous placerons 1 euro supplémentaire de crédit sur le nouveau 1 de poids fort. (Si *max* n'augmente pas, nous pouvons dépenser ce 1 euro superflu.) Comme RÉINITIALISER manipule uniquement les bits situés jusqu'à l'emplacement *A.max* et que chaque bit situé jusqu'à cette position doit être devenu le 1 de poids fort à un certain moment, avant que le 1 de poids fort soit passé à *A.max*, chaque bit vu par RÉINITIALISER a 1 euro de crédit sur lui. Donc la mise à zéro des bits de *A* par RÉINITIALISER peut être complètement payée par le crédit stocké sur les bits. Nous n'avons besoin que de 1 euro pour payer la réinitialisation de *max*.

Il suffit donc de facturer 4 euros pour chaque INCRÉMENTER et 1 euro pour chaque RÉINITIALISER pour que la séquence de n opérations INCRÉMENTER et RÉINITIALISER prenne un temps $O(n)$.

Solutions choisies du chapitre 21 : Structures de données pour ensembles disjoints

Solution de l'exercice 21.2-3

Nous voulons montrer que nous pouvons assigner des charges $O(1)$ à CRÉER-ENSEMBLE et TROUVER-ENSEMBLE et une charge $O(\lg n)$ à UNION, de telle façon que les charges pour une séquence de ces opérations suffisent à couvrir le coût de la séquence, $O(m + n \lg n)$ d'après le théorème. Quand on parle de la charge pour chaque sorte d'opération, cela aide de pouvoir aussi parler du nombre de chaque sorte d'opération.

Considérons la séquence usuelle de m opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE, dont n sont des opérations CRÉER-ENSEMBLE, et soit $l < n$ le nombre d'opérations UNION. (Rappelez-vous la discussion à la section 21.1 sur le fait qu'il y a au plus $n - 1$ opérations UNION.) Il y a alors n opérations CRÉER-ENSEMBLE, l opérations UNION et $m - n - \text{l}$ opérations TROUVER-ENSEMBLE.

Le théorème ne nommait pas séparément le nombre l de UNION; mais bornait plutôt le nombre par n. Si vous reprenez la démonstration du théorème avec l UNION, vous obtenez la borne temporelle $O(m - \text{l} + l \lg l)$ pour la séquence d'opérations. C'est-à-dire que la durée réelle de la séquence d'opérations vaut au plus $c(m + l \lg l)$, pour une certaine constante c.

Par conséquent, nous devons assigner des charges d'opération telles que

(Charge CRÉER-ENSEMBLE)	.	n
+ (Charge TROUVER-ENSEMBLE)	.	$(m - n - \text{l})$
+ (Charge UNION)	.	l
$\geq c(m + l \lg l)$,		

afin que les coûts amortis donnent une borne supérieure pour les coûts réels.

Les assignations suivantes marchent, c' étant une certaine constante $\geq c$:

- CRÉER-ENSEMBLE : c'
- TROUVER-ENSEMBLE : c'
- UNION: $c'(\lg n + 1)$

En substituant dans la somme précédente, nous obtenons

$$\begin{aligned} c'n + c'(m - n - l) + c'(\lg n + 1)l &= c'm + c'l \lg n \\ &= c'(m + l \lg n) \\ &> c(m + l \lg l) \end{aligned}$$

Solution de l'exercice 21.2-6

Appelons les deux listes A et B, et supposons que le représentant de la nouvelle liste soit le représentant de A. Au lieu d'ajouter B à la fin de A, insérons B dans A juste après le premier élément de A. Comme nous devons, de toute façon, traverser B pour actualiser les pointeurs vers l'objet, nous pouvons nous contenter de faire pointer le dernier élément de B vers le deuxième élément de A.

Solutions choisies du chapitre 22 : Algorithmes élémentaires pour les graphes

Solution de l'exercice 22.1-7

$$BB^T(i, j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}$$

- Si $i = j$, alors $b_{ie}b_{je} = 1$ (c'est $1 \cdot 1$ ou $(-1) \cdot (-1)$) chaque fois que e entre dans ou sort du sommet i, et 0 sinon.
- Si $i \neq j$, alors $b_{ie}b_{je} = -1$ quand $e = (i, j)$ ou $e = (j, i)$, et 0 sinon.

Par conséquent,

$$BB^T(i, j) = \begin{cases} \text{degré de } i = \text{degré entrant + degré sortant} & \text{si } i = j. \\ -\text{ (nb. d'arcs reliant } i \text{ et } j) & \text{si } i \neq j. \end{cases}$$

Solution de l'exercice 22.2-5

La démonstration de la validité de l'algorithme PL montre que $u.d = \delta(s, u)$, et l'algorithme ne suppose pas que les listes d'adjacence sont dans quelque ordre particulier que ce soit.
Sur la figure 22.3, si t précède x dans $Adj[w]$, nous pouvons obtenir l'arbre de parcours en largeur montré sur la figure. Mais, si x précède t dans $Adj[w]$ et u précède y dans $Adj[x]$, nous pouvons obtenir l'arc (x, u) dans l'arbre de parcours en largeur.

Solution de l'exercice 22.3-12

Le pseudo-code suivant les procédures PP et VISITER-PP pour assigner des valeurs aux attributs *cc* des sommets.

```
PP(G)
pour chaque sommet u ∈ G.V
    u.couleur = BLANC
    u.π = NIL
date = 0
compteur = 0
pour chaque sommet u ∈ G.V
    si u.couleur == BLANC
            compteur = compteur + 1
            VISITER-PP(G, u, compteur)
```

VISITER-PP(*G*, *u*, *compteur*)
u.cc = *compteur* // étiquetage du sommet
date = *date* + 1
u.d = *date*
u.couleur = GRIS
pour chaque *v* Є *G.Adj*[*u*]
  si *v.couleur* == BLANC
    v.π = *u*
    VISITER-PP(*G*, *v*, *compteur*)
u.couleur = NOIR
date = *date* + 1
u.f = *date*

Cette procédure PP incrémente un compteur chaque fois que l'on appelle VISITER-PP pour faire pousser un nouvel arbre dans la forêt PP. Chaque sommet visité (et ajouté à l'arbre) par VISITER-PP est étiqueté avec cette même valeur de compteur. Donc, *u.cc* = *v.cc* si et seulement si *u* et *v* sont visités dans le même appel à VISITER-PP effectué depuis PP, et la valeur finale du compteur est le nombre d'appels ayant été faits à VISITER-PP depuis PP. En outre, comme en fin de compte chaque sommet est visité, chaque sommet est étiqueté.

Par conséquent, il nous suffit de montrer que les sommets visités par chaque appel à VISITER-PP fait depuis PP correspondent exactement aux sommets d'une même composante connexe de *G*.

- Tous les sommets d'une composante connexe sont visités par un appel à VISITER-PP fait depuis PP :

  Soit *u* le premier sommet de la composante *C* visitée par VISITER-PP. Comme un sommet ne devient non-blanc que quand il est visité, tous les sommets de *C* sont blancs quand on appelle VISITER-PP pour *u*. Donc, d'après le théorème du chemin blanc, tous les sommets de *C* deviennent des descendants de *u* dans la forêt, ce qui signifie que tous les sommets de *C* sont visités (par des appels récursifs à VISITER-PP) avant que VISITER-PP rende la main à PP.

- Tous les sommets visités par un appel à VISITER-PP fait depuis PP sont dans la même composante connexe :

  Si deux sommets sont visités dans le même appel à VISITER-PP depuis PP, ils appartiennent à la même composante connexe, car les sommets sont visités uniquement en suivant des chemins de *G* (en suivant des arcs trouvés dans les listes d'adjacence, à partir de chaque sommet).

Solution de l'exercice 22.4-3

Un graphe non orienté est acyclique (c'est-à-dire, une forêt) si et seulement si un PP ne donne pas d'arcs arrière.

- S'il y a un arc arrière, il y a un cycle.
- S'il n'y a pas d'arc arrière, alors d'après le théorème 22.10, il n'y a que des arcs de liaison. Donc, le graphe est acyclique.

Nous pouvons donc exécuter PP : si nous trouvons un arc arrière, c'est qu'il y a un cycle.

- Durée : $O(V)$. (Pas $O(V + E)$!)

  Si nous voyons $|V|$ arcs distincts, nous avons forcément vu un arc arrière, car (d'après le théorème B.2) dans une forêt (non orientée) acyclique, $|E| \leq |V| - 1$.

Solution du problème 22-1

a. 1. Supposons que (u, v) soit un arc arrière ou avant dans un PL d'un graphe non orienté. Alors, l'un de u et v, disons u, est un ancêtre propre de l'autre (v) dans l'arbre de PL. Comme nous explorons tous les arcs de u avant d'explorer des arcs d'un des descendants de u, nous devons explorer l'arc (u, v) au moment où nous explorons u. Mais alors, (u, v) est forcément un arc de liaison.

2. Dans PL, un arc (u, v) est un arc de liaison quand nous faisons $v.\pi = u$. Mais nous ne le faisons que quand nous faisons $v.d = u.d + 1$. Comme ni $u.d$ ni $v.d$ ne changent jamais ensuite, nous avons $v.d = u.d + 1$ quand PL se termine.

3. Considérons un arc transverse (u, v) où, sans nuire à la généralité, u est visité avant v. Au moment où nous visitons u, le sommet v est forcément déjà dans la file, car sinon (u, v) serait un arc de liaison. Comme v est dans la file, nous avons $v.d \leq u.d + 1$ d'après le lemme 22.3. D'après le corollaire 22.4, nous avons $v.d \geq u.d$. Donc, on a $v.d = u.d$ ou $v.d = u.d + 1$.

b. 1. Supposons que (u, v) soit un arc avant. Alors, nous l'aurions exploré en visitant u et il aurait été un arc de liaison.

2. Idem que pour les graphes non orientés.

3. Pour tout arc (u, v), qu'il s'agisse ou non d'un arc transverse, nous ne pouvons pas avoir $v.d > u.d + 1$, car nous visitons v en dernier quand nous explorons l'arc (u, v). Donc, $v.d \leq u.d + 1$.

4. Visiblement, $v.d \geq 0$ pour tous les sommets v. Pour un arc arrière (u, v), v est un ancêtre de u dans l'arbre PL, ce qui signifie que $v.d \leq u.d$. (Notez que, comme les boucles sont considérées comme des arcs arrière, nous pourrions avoir $u = v$.)

Solutions choisies du chapitre 23 : Arbres couvrants minimaux

Solution de l'exercice 23.1-1

Le théorème 23.1 prouve cela.

Soit A l'ensemble vide et S un ensemble quelconque contenant u mais pas v.

Solution de l'exercice 23.1-4

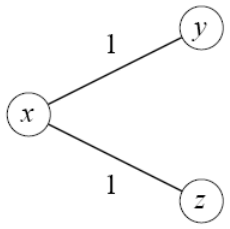
Un triangle dont les poids d'arc sont tous égaux est un graphe dans lequel chaque arc est un arc léger traversant une certaine coupe. Mais, comme le triangle est cyclique, ce n'est pas un arbre couvrant minimum.

Solution de l'exercice 23.1-6

Supposons que, pour toute coupe de G, il existe un arc léger unique traversant la coupe. Considérons deux arbres couvrants minimum, T et T' de G. Nous allons montrer que chaque arc de T est aussi dans T', ce qui signifie que T et T' sont le même arbre et donc qu'il y a un arbre couvrant minimum unique.

Considérons un arc $(u, v) \in T$. Si nous supprimons (u, v) de T, alors T devient non connexe, ce qui donne une coupe $(S, V - S)$. L'arc (u, v) est un arc léger traversant la coupe $(S, V - S)$ (d'après l'exercice 23.1-3). Considérons maintenant l'arc $(x, y) \in T'$ qui traverse $(S, V - S)$. Lui aussi est un arc léger traversant cette coupe. Comme l'arc léger traversant $(S, V - S)$ est unique, les arcs (u, v) et (x, y) sont le même arc. Donc, $(u, v) \in T'$. Comme nous avons choisi (u, v) de façon arbitraire, chaque arc de T est aussi dans T'.

Voici un contre-exemple pour l'inverse :



Ici, le graphe est son propre arbre couvrant minimum, et donc l'arbre couvrant minimum est unique. Considérons la coupe $(\{x\}, \{y, z\})$. Les deux arcs (x, y) et (x, z) sont des arcs légers traversant la coupe et ils sont, tous les deux, des arcs légers.

Solutions choisies du chapitre 24 : Plus courts chemins à origine unique

Solution de l'exercice 24.1-3

Si le nombre maximal d'arcs d'un quelconque plus court chemin partant de la source est m, alors la propriété de relâchement de chemin nous dit que, après m itérations de BELLMAN-FORD, chaque sommet v a réalisé son poids de plus court chemin en $v.d$. D'après la propriété de la borne supérieure, après m itérations, plus aucune valeur d ne changera. Par conséquent, plus aucune valeur d ne changera dans la $(m + 1)$-ème itération. Comme nous ne connaissons pas m à l'avance, nous ne pouvons pas obliger l'algorithme à boucler exactement m fois puis à se terminer. Mais si nous nous contentons de faire en sorte que l'algorithme s'arrête quand plus rien ne change, alors il s'arrêtera après $m + 1$ itérations.

```
BELLMAN-FORD-(M+1)(G, w, s)
SOURCE-UNIQUE-INITIALISATION(G, s)
modifs = VRAI
tant que modifs == VRAI
    modifs = FAUX
    pour chaque arc (u, v) ∈ G.A
            RELÂCHER-M(u, v, w)

RELÂCHER-M(u, v, w)
si v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
    modifs = VRAI
```

Le test d'un cycle de poids négatif (basé sur l'existence d'une valeur d qui changerait si l'on effectuait une autre étape de relâchement) a été supprimé dans le pseudo-code, car cette version de l'algorithme ne sort jamais de la boucle **tant que** à moins que toutes les valeurs d ne cessent de changer.

Solution de l'exercice 24.3-3

Oui, l'algorithme fonctionne encore. Soit u le sommet restant qui n'est pas extrait de la file de priorités F. Si u n'est pas accessible depuis s, alors $u.d = \delta(s, u) = \infty$. Si u est accessible depuis s, alors il y a un plus court chemin $p = s \rightsquigarrow x \rightarrow u$. Quand le nœud x a été extrait, $x.d = \delta(s, u)$ puis l'arc (x, u) a été relâché ; donc, $u.d = \delta(s, u)$.

Solution de l'exercice 24.3-6

Pour trouver le chemin le plus fiable entre s et t, exécutez l'algorithme de Dijkstra avec les poids d'arc $w(u, v) = -\lg r(u, v)$ pour trouver les plus courts chemins depuis s en temps $O(A + S \lg S)$. Le chemin le plus fiable est le plus court chemin de s à t, et la fiabilité de ce chemin est le produit des fiabilités de ses arcs.

Voici pourquoi cette méthode fonctionne. Comme les probabilités sont indépendantes, la probabilité qu'un chemin ne s'effondrera pas est le produit des probabilités que ses arcs ne s'effondreront pas. Nous voulons trouver un chemin $s \overset{p}{\rightsquigarrow} t$ tel que $\Pi_{(u, v) \in p} r(u, v)$ soit maximisé. Cela équivaut à maximiser $\lg (\Pi_{(u, v) \in p} r(u, v)) = \sum_{(u, v) \in p} \lg r(u, v)$, ce qui revient à minimiser $\sum_{(u, v) \in p} - \lg r(u, v)$. (Remarque : $r(u, v)$ peut valoir 0 et $\lg 0$ être donc indéfini. Dans cet algorithme, définissons donc $\lg 0 = -\infty$.) Par conséquent, si nous assignons les poids $w(u, v) = - \lg r(u, v)$, nous avons un problème de plus court chemin.
Comme $\lg 1 = 0$, $\lg x < 0$ pour $0 < x < 1$ et que nous avons défini $\lg 0 = -\infty$, tous les poids w sont non négatifs et nous pouvons employer l'algorithme de Dijkstra pour trouver les plus courts chemins depuis s en temps $O(A + S \lg S)$.

Autre solution

Vous pouvez aussi travailler avec les probabilités originales en exécutant une version modifiée de l'algorithme de Dijkstra qui maximise le produit des fiabilités sur un chemin au lieu de minimiser la somme des poids sur un chemin.

Dans l'algorithme de Dijkstra, utilisez les fiabilités comme poids d'arc et substituez :

- max (et EXTRAIRE-MAX) à min (et EXTRAIRE-MIN) dans le relâchement et dans la file,
- . à + dans le relâchement,
- 1 (élément neutre pour .) à 0 (élément neutre pour +) et $-\infty$ (élément neutre pour min) à ∞ (élément neutre pour max).

Par exemple, nous devrions employer le code suivant au lieu de la procédure RELÂCHER usuelle :

RELÂCHER-FIABILITÉ(u, v, r)
si $v.d < u.d \,.\, r(u, v)$
  $v.d = u.d \,.\, r(u, v)$
  $v.\pi = u$

Cet algorithme est isomorphe au précédent : il effectue les mêmes opérations, sauf qu'il travaille avec les probabilités originelles au lieu des probabilités transformées.

Solution de l'exercice 24.4-7

Observez que, après la première passe, toutes les valeurs d sont au plus 0 et que le relâchement des arcs (v_0, v_i) ne modifiera jamais une valeur d. Par conséquent, nous pouvons éliminer v_0 en exécutant l'algorithme de Bellman-Ford sur le graphe des contraintes sans le nœud v_0, mais en initialisant toutes les estimations de plus court chemin à 0 au lieu de 1.

Solution de l'exercice 24.5-4

Chaque fois que RELÂCHER initialise π pour un certain sommet, la procédure diminue aussi la valeur d du sommet. Si donc $s.\pi$ est initialisé à une valeur non NIL, $s.d$ passe de sa valeur initiale 0 à une valeur négative. Mais $s.d$ est le poids d'un certain chemin de s à s, qui est un cycle contenant s. Donc, il existe un cycle de poids négatif.

Solution du problème 24-3

a. Nous pouvons employer l'algorithme de Bellman-Ford sur un graphe orienté pondéré idoine, $G = (S, A)$, que nous allons former comme suit. Il y a un sommet dans S pour chaque devise et, pour chaque paire de devises c_i et c_j, il y a des arcs orientés (v_i, v_j) et (v_j, v_i). (Donc, $|S| = n$ et $|A| = n(n-1)$.)

Pour déterminer les poids d'arc, commençons par observer que :

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

si et seulement si

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1$$

En prenant les logarithmes des deux côtés de l'inégalité précédente, nous exprimons cette condition sous la forme

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \cdots + \lg \frac{1}{R[i_{k-1}, i_k]} + \lg \frac{1}{R[i_k, i_1]} < 0$$

Par conséquent, si nous définissons le poids de l'arc (v_i, v_j) comme

$$\begin{aligned} w(v_i, v_j) &= \lg \frac{1}{R[i, j]} \\ &= -\lg R[i, j] \end{aligned}$$

alors nous devons déterminer s'il existe un cycle de poids négatif dans G ayant ces poids d'arc.

Nous pouvons déterminer s'il existe un cycle de poids négatif dans G en ajoutant un sommet supplémentaire v_0 avec des arcs de poids 0 (v_0, v_i) pour tout $v_i \in S$, en exécutant BELLMAN-FORD depuis v_0 et en utilisant le résultat booléen de BELLMAN-FORD (qui est VRAI s'il n'y a pas de cycles de poids négatif et FAUX s'il y a un cycle de poids négatif) pour guider notre réponse. C'est-à-dire que nous inversons le résultat booléen de BELLMAN-FORD.

Cette méthode fonctionne parce que l'ajout du nouveau sommet v_0 avec des arcs de poids 0 reliant v_0 à tous les autres sommets ne peut pas créer de nouveaux cycles, tout en garantissant que tous les cycles de poids négatif sont accessibles depuis v_0.

Il faut un temps $\Theta(n^2)$ pour créer G, qui a $\Theta(n^2)$ arcs. Ensuite, il faut un temps $O(n^3)$ pour exécuter BELLMAN-FORD. Donc, le temps total est $O(n^3)$.

Une autre façon de déterminer s'il existe un cycle de poids négatif consiste à créer G et, sans ajouter v_0 et ses arcs incidents, à exécuter l'un des algorithmes de plus courts chemins toutes paires. Si la matrice des distances de plus court chemin résultante a des valeurs négatives sur la diagonale, alors il y a un cycle de poids négatif.

b. En supposant que nous exécutions BELLMAN-FORD pour résoudre la partie (a), il nous suffit de trouver les sommets d'un cycle de poids négatif. Nous pouvons procéder comme suit. Primo, nous relâchons tous les arcs une fois de plus. Comme il y a un cycle de poids négatif, la valeur d d'un certain sommet u changera. Tout ce dont nous avons besoin, c'est de suivre de façon répétée les valeurs π jusqu'à revenir en u. Autrement dit, nous pouvons employer la méthode récursive donnée par la procédure AFFICHER-CHEMIN de la section 22.2, mais nous arrêtons quand elle revient au sommet u.

La durée d'exécution est $O(n^3)$ pour exécuter BELLMAN-FORD, plus $O(n)$ pour afficher les sommets du cycle, soit en tout un temps $O(n^3)$.

Solutions choisies du chapitre 25 : Plus courts chemins toutes paires

Solution de l'exercice 25.1-3

La matrice $L^{(0)}$ correspond à la matrice identité

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

de la multiplication matricielle ordinaire. Substituez 0 (l'identité pour +) à ∞ (l'identité pour min) et 1 (l'identité pour $\cdot$) à 0 (l'identité pour +).

Solution de l'exercice 25.1-5

L'algorithme des plus courts chemins toutes paires de la section 25.1 calcule
$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1}$
où $l_{ij}^{(n-1)} = \delta(i, j)$ et $L^{(0)}$ est la matrice identité. C'est-à-dire que l'élément à la *i*ème ligne et la *j*ème colonne du « produit » matriciel est la distance de plus court chemin du sommet *i* au sommet *j*, et la ligne *i* du produit est la solution du problème des plus courts chemins à origine unique pour le sommet *i*.

Remarquez que, dans un « produit » matriciel C = A · B, la *i*ème ligne de C est la *i*ème ligne de A « multipliée » par B. Comme tout ce que nous voulons, c'est la *i*ème ligne de C, nous n'avons jamais besoin de plus que la *i*ème ligne de A.
Par conséquent, la solution du problème des plus courts chemins à origine unique depuis le sommet *i* est $L_i^{(0)} \cdot W^{n-1}$, où $L_i^{(0)}$ est la *i*ème ligne de $L^{(0)}$ (vecteur dont le *i*ème élément est 0 et les autres sont ∞).
Effectuer les « multiplications » précédentes en partant de la gauche revient essentiellement au même que l'algorithme BELLMAN-FORD. Le vecteur correspond aux valeurs *d* de BELLMAN-FORD, estimations des plus courts chemins entre l'origine et chaque sommet.

- Le vecteur vaut initialement 0 pour l'origine et ∞ pour tous les autres sommets, la même chose que pour les valeurs configurées pour *d* par SOURCE-UNIQUE-INITIALISATION.

- Chaque « multiplication » du vecteur courant par W relâche tous les arcs, tout comme le fait BELLMAN-FORD. C'est-à-dire qu'une estimation de distance dans la ligne, disons la distance vers v, est remplacée par une distance inférieure, s'il y en a une, formée par ajout d'un certain $w(u, v)$ à l'estimation courante de la distance à u.
- Le relâchement/multiplication se fait $n - 1$ fois.

Solution de l'exercice 25.2-4

Avec les exposants, le calcul est $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$. Si, ayant laissé tomber les exposants, nous devions calculer et mémoriser d_{ik} ou d_{kj} avant d'employer ces valeurs pour calculer dij, il se pourrait que nous calculions l'une des valeurs suivantes :
$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)})$
$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)})$
$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)})$
Dans n'importe lequel de ces scénarios, nous calculons le poids d'un plus court chemin entre i et j avec tous les sommets intermédiaires dans $\{1, 2, \ldots, k\}$. Si nous employons $d_{ik}^{(k)}$, et non $d_{ik}^{(k-1)}$, dans le calcul, alors nous utilisons un sous-chemin entre i et k avec tous les sommets intermédiaires dans $\{1, 2, \ldots, k\}$. Mais k ne peut pas être un sommet *intermédiaire* sur un plus court chemin de i à k, car sinon il y aurait un cycle sur ce plus court chemin. Donc, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. Un raisonnement similaire montre que $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Par conséquent, nous pouvons laisser tomber les exposants dans le calcul.

Solution de l'exercice 25.3-4

Cela modifie les plus courts chemins. Soit le graphe $V = \{s, x, y, z\}$, avec 4 arcs : $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$ et $w(s, z) = -10$. Ajoutons donc 10 à chaque poids pour faire $\widehat{w}$. Avec $\widehat{w}$, le plus court chemin entre s et y est $s \to x \to y$, avec le poids 4. Avec $\widehat{w}$, le plus court chemin entre s et y est $s \to y$, avec le poids 15. (Le chemin $s \to x \to y$ a le poids 24.) Le problème est que, si l'on se contente d'ajouter la même quantité à chaque arc, on pénalise les chemins ayant plus d'arcs, même si leurs poids sont faibles.

Solutions choisies du chapitre 26 : Flot maximal

Solution de l'exercice 26.2-11

Pour deux sommets quelconques u et v de G, nous pouvons définir un réseau de flot G_{uv} composé de la version orientée de G avec $s = u$, $t = v$ et toutes les capacités d'arc fixées à 1. (Le réseau de flot G_{uv} a S sommets et $2|A|$ arcs, de sorte qu'il possède $O(S)$ sommets et $O(A)$ arcs, comme exigé. Il faut que toutes les capacités soient 1, de façon que le nombre d'arcs de G traversant une coupe soit égal à la capacité de la coupe dans G_{uv}.) Soit f_{uv} un flot maximum dans G_{uv}.

Nous affirmons que, pour tout $u \in S$, la connectivité d'arc k est égale à $\min_{v \in S-\{u\}} \{|f_{uv}|\}$.

Nous prouverons plus loin la validité de cette affirmation. En supposant qu'elle soit vraie, nous pouvons trouver k ainsi :

CONNECTIVITÉ-D'ARC(G)
$k = \infty$
choisir un sommet $u \in G.S$
pour chaque sommet $v \in G.S - \{u\}$
    configurer le réseau de flot G_{uv} comme précédemment expliqué
    trouver le flot maximal f_{uv} sur G_{uv}
    $k = \min(k, |f_{uv}|)$
retourner k

L'affirmation découle du théorème du flot maximal/coupe minimale et de la façon dont nous avons choisi les capacités pour que la capacité d'une coupe soit le nombre d'arcs la traversant. Nous allons montrer que $k = \min_{v \in S-\{u\}} \{|f_{uv}|\}$ pour tout $u \in S$ en montrant séparément que k est au moins ce minimum et que k est au plus ce minimum.

- Démonstration que $k \geq \min_{v \in S-\{u\}} \{|f_{uv}|\}$. Soit $m = \min_{v \in S-\{u\}} \{|f_{uv}|\}$.

Supposez que nous supprimions seulement $m - 1$ arcs de G. Pour tout sommet v, d'après le théorème du flot maximal/coupe minimale, u et v sont encore connectés. (Le flot max de u vers v est au moins m, donc toute coupe séparant u de v a une capacité au moins m, ce qui implique qu'au moins m arcs traversent une telle coupe. Par conséquent, il reste au moins un arc traversant la coupe quand nous supprimons $m - 1$ arcs.) Par conséquent, chaque nœud est connecté à u, ce qui implique que le graphe est encore connexe. Il faut donc supprimer au moins m arcs pour que le graphe ne soit plus connexe, ce qui veut dire que $k \geq \min_{v \in S-\{u\}} \{|f_{uv}|\}$

- Démonstration que $k \leq \min_{v \in S-\{u\}} \{|f_{uv}|\}$

Considérons un sommet v ayant le $|f_{uv}|$ minimum. D'après le théorème du flot maximal/coupe minimale, il y a une coupe de capacité $|f_{uv}|$ séparant u et v. Comme toutes les capacités d'arc sont 1, il y a exactement $|f_{uv}|$ arcs qui traversent cette coupe. Si ces arcs sont supprimés, il n'y a pas de chemin entre u et v, de sorte que notre graphe devient non connexe. D'où $k \leq \min_{v \in S-\{u\}} \{|f_{uv}|\}$.

- En conséquence, l'affirmation que $k \leq \min_{v \in S-\{u\}} \{|f_{uv}|\}$ pour tout $u \in S$ est vraie.

Solution de l'exercice 26.3-3

Par définition, un chemin améliorant est un chemin simple $s \rightsquigarrow t$ du réseau résiduel G'_f. Comme G n'a pas d'arcs entre des sommets de L ni d'arcs entre des sommets de R, c'est aussi le cas du réseau de flot G' et donc de G'_f. De plus, les seuls arcs impliquant s ou t connectent s à L et R à t. Remarquez que, même si les arcs de G' ne peuvent aller que de L à R, les arcs de G'_f peuvent aussi aller de R à L.

Par conséquent, tout chemin améliorant doit suivre

$s \rightarrow L \rightarrow R \rightarrow \ldots \rightarrow L \rightarrow R \rightarrow t$,

circulant, dans un sens puis dans l'autre, entre L et R au plus autant de fois qu'il peut le faire sans utiliser deux fois un sommet. Il contient s, t et des nombres égaux de sommets distincts de L et R (au plus $2 + 2 \cdot \min(|L|, |R|)$ sommets en tout). La longueur d'un chemin améliorant (à savoir, son nombre d'arcs) est donc bornée supérieurement par $2 \cdot \min(|L|, |R|)$.

Solution du problème 26-4

a. Il suffit d'exécuter une itération de l'algorithme de Ford-Fulkerson. L'arc (u, v) dans A avec capacité améliorée garantit que l'arc (u, v) est dans le réseau résiduel. Recherchez donc un chemin améliorant et, si vous en trouvez un, actualisez le flot.

Durée

$O(S + A) = O(A)$ si nous trouvons le chemin améliorant par une recherche en profondeur ou en largeur.
Pour voir pourquoi il suffit d'une seule itération, considérons séparément les cas où (u, v) est ou n'est pas un arc qui traverse une coupe minimum. Si (u, v) ne traverse pas de coupe minimum, alors augmenter sa capacité ne modifie pas la capacité d'une quelconque coupe minimum, et donc la valeur du flot maximum ne change pas. Si (u, v) traverse une coupe minimum, alors augmenter sa capacité de 1 augmente la capacité de cette coupe de 1, et donc risque d'augmenter la valeur du flot maximum de 1. Dans ce cas, soit il n'y a pas de chemin améliorant (auquel cas c'est qu'il y avait une autre coupe minimum que (u, v) ne traverse pas), soit le chemin améliorant augmente le flot de 1. Dans tous les cas, une itération de Ford-Fulkerson suffit.

b. Soit f le flot maximum avant de diminuer $c(u, v)$.

Si $f(u, v) = 0$, il n'y a pas besoin de faire quoi que ce soit.
Si $f(u, v) > 0$, il faut actualiser le flot maximum. Supposons désormais que $f(u, v) > 0$, et donc que $f(u, v) \geq 1$.

Soit $f'(x, y) = f(x, y)$ pour tous $x, y \in S$, sauf que $f'(u, v) = f(u, v) - 1$. Bien que f' obéisse à toutes les contraintes de capacité, même après diminution de $c(u, v)$, ce n'est pas un flot licite, car il enfreint la conservation de flot en u (sauf si $u = s$) et en v (sauf si $v = t$). f' a une unité de flot de plus en entrée qu'en sortie en u et une unité de flot de plus en sortie qu'en entrée en v.

L'idée est de rediriger cette unité de flot, de façon qu'elle sorte de u et arrive en v par un autre chemin. Si c'est impossible, nous devons diminuer le flot de s à u et de v à t d'une unité.

Recherchons un chemin améliorant de u à v (remarque : *pas* de s à t).

- S'il existe un tel chemin, on augmente le flot sur ce chemin.
- S'il n'existe pas de tel chemin, on diminue le flot de s à u en augmentant le flot de u à s. C'est-à-dire, on trouve un chemin améliorant $u \rightsquigarrow s$ et on augmente le flot sur ce chemin. (Un tel chemin existe à coup sûr, car il y a du flot entre s et u.) De même, on diminue le flot de v à t en trouvant un chemin améliorant $t \rightsquigarrow v$ et en augmentant le flot sur ce chemin.

Durée

$O(S + A) = O(A)$ si nous trouvons le chemin améliorant par une recherche en profondeur ou en largeur.