INFOF-203 – Syllabus d'exercices

Jean Cardinal, Justin Dallant & Robin Petit

Remerciements à Axel Abels, Gian Marco Paldino et Sarah Van Bogaert

2021-2022

Table des matières

Séance 1 — Échauffement

Rappel (Équivalence asymptotique)**.** *Pour deux fonctions* $f, g : \mathbb{N} \to \mathbb{R}$ *telles que :*

$$\exists N > 0 \text{ s.t. } \forall n \geq N : g(n) \neq 0,$$

on dit que f *et* g *sont équivalents asymptotiquement lorsque :*

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 1,$$

et que l'on note $f \sim g$*.*

Exercice 1.1. Écrire un programme qui, étant donné un tableau `a[]` de n entiers distincts, trouve un *minimum local* : un indice `i` compris entre 0 et $n-1$ tel que `a[i]<a[i+1]` et `a[i]<a[i-1]` (à condition que les bornes existent). Le programme ne devra effectuer que $\sim 2 \log n$ comparaisons au pire cas.

Résolution. Il est facile de se convaincre qu'un tel minimum local existe toujours. En effet, puisque tous les éléments du vecteur sont distincts, `a` admet un unique minimum, et ce dernier est en particulier un minimum local.

Pour le trouver, on procède sous forme de recherche dichotomique :

```
public int localMinimum(Comparable[] a) {
  int hi = a.length-1;
  int lo = 0;
  int m;
  if(hi == lo || a[0].compareTo(a[1]) < 0)
    return 0;
  if(a[hi].compareTo(a[hi-1]) < 0)
    return hi;
  while(hi != lo) {
    m = (hi+lo)/2;
    if(a[m-1].compareTo(a[m]) > 0) {
      if(a[m].compareTo(a[m+1]) > 0)
        lo = m+1; // a[m-1] > a[m] > a[m+1]
      else
        return m; // a[m-1] > a[m] and a[m] < a[m+1]
    } else {
      if(a[m].compareTo(a[m+1]) > 0)
        lo = m+1; // a[m-1] < a[m] and a[m] > a[m+1]
      else
        hi = m-1; // a[m-1] < a[m] < a[m+1]
    }
  }
  return hi;
}
```

La recherche se fait donc en $O(\log n)$ comparaisons dans le pire des cas car à chaque étape la taille du sous-vecteur d'intérêt en divisé par 2. De plus à chaque étape, deux comparaisons sont à faire : `a[m] < a[m-1]` et `a[m] < a[m+1]`. Il y a donc $2 \log_2 n + 2 = 2 \log_2 n + o(\log_2 n) \sim 2 \log_2 n$ comparaisons effectuées dans le pire des cas.

Montrons alors que l'algorithme trouve bien un minimum local. Faisons cela par récurrence. Si $n = 1$, alors il est trivial que $m = 0$ est bien l'indice d'un minimum local. Supposons maintenant que l'algorithme trouve un minimum local pour tout vecteur de taille $k \lneq n$ et montrons qu'il trouve un minimum local pour tout vecteur de taille n. Fixons $m = \lfloor n/2 \rfloor$. Si `a[m]` est un minimum local, alors la démonstration est finie. Si par contre `a[m]` n'est pas un minimum local, alors soit `a[m] > a[m+1]` soit `a[m] > a[m-1]` (possiblement les deux à la fois). Considérons uniquement le premier cas car le second se traite de manière symétrique. Le sous-vecteur `a[m+1:n-1]` un vecteur de $n - m - 1 < n$ éléments distincts. Par hypothèse de récurrence, on sait qu'il existe un indice k tel que `a[k]` est un minimum local de `a[m+1:n-1]`. Distinguons les trois cas suivants :

1. si $m + 1 < k < n - 1$, alors en particulier k est un minimum local de `a` ;
2. si $k = m + 1$, alors k est le premier élément du sous-vecteur `a[m+1:n-1]`, et donc on sait que `a[m+1] < a[m+2]`. Or on sait également que `a[m] > a[m+1]`. On en déduit que k est un minimum local de `a` ;
3. finalement si $k = n - 1$, alors k est le dernier élément du sous-vecteur `a[m+1:n-1]` et on sait que `a[n-1] < a[n-2]`. k est dès lors un minimum local du vecteur `a`.

□

Exercice 1.2. Un tableau `a[]` est dit *bitonique* s'il existe un indice `i` tel que les éléments d'indices entre 0 et `i` sont triés en ordre strictement croissant, et ceux d'indices supérieurs à `i` sont triés en ordre strictement décroissant. Écrire un programme qui détermine si une valeur donnée appartient au tableau. Ce programme ne devra effectuer que $\sim 3 \log n$ comparaisons au pire cas.

Résolution. Tout d'abord, voici une fonction récursive qui permet de trouver l'indice du maximum dans un tableau bitonique.

```
public static int max(int[] a, int lo, int hi) {
    if (hi == lo) return hi;
    int mid = (lo + hi) / 2;
    if (a[mid] < a[mid + 1]) return max(a, mid+1, hi);
    else return max(a, lo, mid);
}
```

Comme à chaque étape, on divise la taille du tableau à considérer par deux, la recherche du maximum se fait en $\sim \log n$ comparaisons.

Lorsqu'on connait l'indice du maximum, on peut séparer le tableau en une partie croissante et une partie décroissante. Sur un tel sous-tableau, on peut utiliser la fonction décrite ci-dessous afin de vérifier la présence d'un élément (`x`) dans ce tableau (`a`). (Notez que le booléen `increasing` précise si la partie considérée du tableau `a` est croissante ou décroissante.)

```
public static boolean appearsIn(int[] a, int x, int lo, int hi, boolean increasing) {
        if (lo > hi) return false;
        else if (lo == hi) return (a[lo] == x);
        int mid = (lo + hi)/2;
        if (a[mid] == x) return true;
        else if ((a[mid] < x) ^ increasing) {
                return appearsIn(a, x, lo, mid-1, increasing);
        } else {
                return appearsIn(a, x, mid+1, hi, increasing);
        }
}
```

Considérons plus en détails l'expression `((a[mid] < x) ^ increasing)`. C'est une forme compacte utilisant un XOR qui permet de distinguer 4 cas. Par exemple, un premier cas est lorsque `(a[mid] < x)` et le tableau est croissant : il faut dans ce cas appeler la fonction récursive `appearsIn` sur le tableau `a` entre l'index `mid+1` et `hi`. Ici, une difficulté se présente. Que se passe-t-il si `mid == lo`? Ceci peut arriver si `hi = lo+1`. Dans ce cas `mid-1 < lo` et le sous-tableau en question n'a aucun sens. C'est pourquoi nous testons l'expression `(lo > hi)` au début du programme. Au cas où `(a[mid] < x)` et le tableau est décroissant, il faut considérer le sous-tableau `a` entre l'index `lo` et `mid-1`. Il est facile de voir que, pour les deux autres cas, le bon sous-tableau est considéré, grâce à l'expression XOR.

La fonction `main` peut s'écrire de la manière suivante :

```
public static void main(String[] args) {
        int[] a = {2, 3, 5, 8, 6, 4, 2};
        int size = a.length;
        int x = 7;
        int maxIna = max(a, 0, size-1);
        boolean inIncreasing = appearsIn(a, x, 0, maxIna, true);
        boolean inDecreasing = appearsIn(a, x, maxIna, size-1, false);
        boolean inArray = (inIncreasing || inDecreasing);
        System.out.println(x + " is in array: " + inArray);
}
```

Nous avons vu que la procédure trouvant le maximum dans le tableau utilisait $\sim \log n$ comparaisons. De même, un appel de la fonction `appearsIn` sur un tableau de taille m utilise $\log m$ comparaisons puisqu'on considère à chaque appel récursif un tableau deux fois plus petit. Comme nous appelons cette fonction au départ sur deux parties (une croissante et une décroissante) du tableau `a`, chacune de ces deux parties a une taille inférieure ou égale à n. Il en découle que le nombre total de comparaisons dans l'algorithme est $\sim 3 \log n$.

□

Exercice 1.3 (Lancer d'oeufs)**.** On dispose d'un immeuble de N étages et de beaucoup d'oeufs. Un oeuf ne se casse que s'il est lancé d'un étage supérieur à F. Donner un algorithme pour déterminer F en lançant au plus $\sim \log N$ oeufs. Dans l'hypothèse où F est beaucoup plus petit que N, donner un algorithme qui lance au plus $\sim 2 \log F$ oeufs.

Résolution. La classe `EggThrow` ci-dessous permet de déterminer F. Une première méthode est implémentée par la fonction récursive `checkFloors`. Il est facile de voir que cette méthode ne comporte que $\sim \log N$ opérations car on diminue à chaque itération le nombre d'étages à considérer.

Une deuxième méthode, implémentée par la fonction `checkFloorsWhenSmall`, est plus rapide lorsque F est beaucoup plus petit que N. Elle consiste d'abord à trouver les puissances de 2 consécutives entre lesquelles se trouve la valeur de F. Cette première étape nécessite exactement $\lceil \log F \rceil$ opérations. Ensuite, la méthode utilise la fonction `checkFloors` pour affiner la recherche. Le nombre d'étages considérés dans cette deuxième étape est $2^{\lceil \log F \rceil - 1}$ [1]. Dès lors, l'étape utilise $\sim \log F$ opérations. Globalement, la deuxième méthode utilise donc $\sim 2 \log F$ opérations.

1. En effet, les deux puissances de 2 consécutives entourant F sont $2^{\lceil \log F \rceil - 1}$ et $2^{\lceil \log F \rceil}$

```
    public int checkFloors(int lo, int hi) {
        int mid = (lo + hi) / 2;
        if (lo == hi) return lo;
        if (eggBroken(mid))
            return checkFloors(lo, mid);
        else
            return checkFloors(mid+1,hi);
    }


    public int checkFloorsWhenSmall(int N) {
        // d'abord, trouver les puissances de 2 consécutives entre
        // lesquelles se trouve la valeur de F
        int floor = 2;
        while (floor < N) {
            if (eggBroken(floor)) break;
            floor *= 2;
        }
        // ensuite, affiner la recherche pour trouver la valeur de F
        return checkFloors(floor/2 + 1, Math.min(floor,N));
    }

    public static void main(String[] argv) {
        int N = 1000;
        //int floorF = myEggThrow.checkFloors(0, N);
        int floorF = myEggThrow.checkFloorsWhenSmall(N);
        System.out.println("The floor F is : " + floorF);
    }
```

□

Exercice 1.4 (Majorité)**.** Donner un algorithme de complexité $O(n \log n)$ qui détermine si un tableau de n éléments deux à deux comparables en temps constant contient un élément majoritaire, c'est-à-dire un élément qui apparaît plus de $\lfloor n/2 \rfloor$ fois.

Résolution. Supposons qu'on ait le tableau suivant :

$$[\text{"}c3\text{"}, \text{"}c2\text{"}, \text{"}c1\text{"}, \text{"}c2\text{"}, \text{"}c4\text{"}, \text{"}c2\text{"}, \text{"}c2\text{"}]$$

Un tri nous donne le tableau :

$$[\text{"}c1\text{"}, \text{"}c2\text{"}, \text{"}c2\text{"}, \text{"}c2\text{"}, \text{"}c2\text{"}, \text{"}c3\text{"}, \text{"}c4\text{"}]$$

On peut ensuite parcourir ce tableau et s'arrêter dès qu'on trouve une séquence d'éléments identiques de longueur $\lfloor n/2 \rfloor + 1 = 4$.

$$[\text{"}c1\text{"}, \text{"}\mathbf{c2}\text{"}, \text{"}\mathbf{c2}\text{"}, \text{"}\mathbf{c2}\text{"}, \text{"}\mathbf{c2}\text{"}, \text{"}c3\text{"}, \text{"}c4\text{"}]$$

Si aucune séquence de cette longueur est trouvée, il n'y a pas d'élément majoritaire dans le tableau.

```
import java.util.Arrays;
public boolean hasMajority(Comparable[] a) {
    Arrays.sort(a);
```

```
4
5       // Recherche d'une sequence de plus de floor(n/2) elements identiques consecutifs
6       Comparable base = a[0];
7       int c = 1;
8       int n = a.length;
9       int i = 1;
10      while (i<n && c<=n/2) {
11          if (a[i]==base) {
12              c++;
13          } else {
14              c = 1;
15              base = a[i];
16          }
17          i++;
18      }
19      return c > n/2;
20  }
```

Il reste a démontrer que la complexité de cet algorithme est bien $O(n \log n)$.

On commence par rappeler que le tri se fait en $O(n \log n)$, puisqu'on suppose qu'une comparaison se fait en temps constant. Toutes les autres opérations se font en un temps constant. Les opérations des lignes 11 à 17 sont cependant répétées un nombre de fois qui dépendra de l'entrée. Au pire des cas on effectue n itérations de cette boucle (si aucun élément majoritaire n'est trouvé). Étant donné le coût constant par itération on a donc une complexité totale de $O(n)$ pour cette boucle.

L'ensemble se simplifie donc en $O(n \log n)$.

Alternativement, on peut prendre l'élément d'indice $\lceil n/2 \rceil$ après le tri et compter le nombre d'occurrences.

□

Exercice 1.5 (Majorité)**.** Donner un algorithme de complexité linéaire pour le problème précédent, qui n'utilise qu'un espace supplémentaire de taille constante, et qui ne compare les éléments que pour l'égalité.

Résolution. Notons tout d'abord qu'on peut vérifier si un candidat est majoritaire en comptant le nombre d'occurrences par un simple parcours, ce qui encourt une complexité linéaire. Pour compléter la réponse il nous faut donc en plus un algorithme qui retourne l'élément majoritaire s'il existe et sinon retourne un élément quelconque.

On remarque tout d'abord que si l'élément e est majoritaire et $f \neq e$ est un autre élément du tableau, e reste majoritaire si on retire une instance de e et de f du tableau. En effet, notons par $\mathrm{count}(A, e)$ le nombre d'occurrences de e dans A. Si e est majoritaire, on sait que $\mathrm{count}(A, e) > |A|/2$. On obtient A' en retirant une instance de e et une instance de f de A. On a alors que $\mathrm{count}(A, e) = \mathrm{count}(A', e) + 1 > |A|/2$ et puisque $|A| = |A'| + 2$, on a $\mathrm{count}(A', e) + 1 > |A'|/2 + 1$. Il suit donc que $\mathrm{count}(A', e) > |A'|/2$, en d'autre termes, e est également majoritaire dans A'.

Puisqu'on est restreint à un espace supplémentaire constant, on souhaite exploiter cette propriété sans faire de copie de notre tableau.

Pour ce faire, on va parcourir le tableau en maintenant à tout moment un élément candidat et un compteur. Lorsqu'un élément identique au candidat est rencontré on incrémente le compteur. Si au contraire un élément différent du candidat est rencontré, on décrémente le compteur. À chaque fois que le compteur est mis à zéro

on a *neutralisé* le candidat, c'est-à-dire qu'on a trouvé autant d'occurrences d'éléments différents du candidat que d'éléments identiques au candidat. Cela nous indique que le candidat n'est pas majoritaire dans le sous-tableau qu'on a déjà parcouru. Si le candidat est majoritaire sur l'ensemble du tableau il devra donc être majoritaire dans la suite du tableau qu'on n'a pas encore parcouru. On peut donc 'ignorer' ce qu'on a parcouru auparavant et ré-initialiser le candidat et le compteur à partir du prochain élément.

En répétant ces actions jusqu'à la fin du tableau on se retrouve à la fin avec un candidat qui est l'élément majoritaire s'il en existe un, et un élément quelconque sinon. Comme argumenté précédemment, on peut vérifier si ce candidat est effectivement majoritaire en comptant le nombre d'occurrences.

On obtient donc l'algorithme suivant :

```java
import java.util.Arrays;
public boolean hasMajority(Comparable[] a) {

    // Recherche du candidat
    Comparable candidat;
    int compteur = 0;
    int i=0;
    int n = a.length;
    while (i<n){
        if (compteur==0){
            candidat = a[i];
        }
        if (a[i]==candidat){
            compteur++;
        } else {
            compteur--;
        }
        i++;
    }
    // Verification du candidat
    compteur = 0;
    i=0;
    while (i<n) {
        if (a[i]==candidat){
            compteur++;
        }
        i++;
    }
    return compteur > n/2;
}
```

En termes de complexité, on a deux parcours du tableau où chaque itération se fait en temps constant. Puisque ces boucles se font séquentiellement leur complexité est additionnée pour obtenir $O(2n) = O(n)$ L'espace supplémentaire ne dépend pas de l'entrée, en d'autres termes, il est constant.

On démontre maintenant de façon informelle que la valeur de `candidat` après la première boucle est bien l'élément majoritaire s'il en existe un.

Supposons qu'il existe un élément majoritaire m. Cet élément est donc présent au moins $\lfloor n/2 \rfloor + 1$ fois dans le tableau. On distingue 4 cas :

1. `candidat` == m et `a[i]` == $m \Rightarrow$ `compteur` est incrémenté

2. `candidat` != m et `a[i]` == $m \Rightarrow$ `compteur` est décrémenté
3. `candidat` == m et `a[i]` != $m \Rightarrow$ `compteur` est décrémenté
4. `candidat` != m et `a[i]` != $m \Rightarrow$ `compteur` est décrémenté ou incrémenté

Notons que comme m est majoritaire, ces deux premiers cas seront également majoritaires. Dans la majorité des cas on augmente donc l'avantage du candidat m ou on réduit l'avantage d'un autre `candidat`$\neq m$. Il suit donc qu'on termine dans un état où `candidat`$= m$ et `compteur` > 0.

Une preuve plus complète est donnée dans l'article ayant introduit l'algorithme Boyer–Moore de vote majoritaire [2] ou la preuve plus récente de Wim H. Hesselink [3].

□

Exercice 1.6. Décrire un algorithme de complexité $O(n)$ pour le problème suivant : étant donné une matrice `a` $n \times n$ dont les éléments apparaissent en ordre croissant dans chaque ligne et chaque colonne, déterminer si un élément `x` donné en entrée appartient à la matrice. On suppose que tous les éléments sont distincts.

Résolution. Afin de déterminer si `x` appartient à la matrice `a`, on va la parcourir depuis le coin inférieur gauche jusqu'au coin supérieur droit en utilisant uniquement les déplacements vers la droite et vers le haut.

Pour s'en convaincre, il faut remarquer que si `a[i][j]` > `x` et `a[i-1][j]` < `x`, alors le couple (i, j) *sépare* la matrice `a` en 4 sous-matrices :

$$a = \left[\begin{array}{c|c} a^{(1)} & a^{(2)} \\ \hline a^{(3)} & a^{(4)} \end{array}\right] = \left[\begin{array}{ccc|ccc} a_{0,0} & \cdots & a_{0,j} & a_{0,j+1} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{i-1,0} & \cdots & a_{i-1,j} & a_{i-1,j+1} & \cdots & a_{i-1,n-1} \\ \hline a_{i,0} & \cdots & a_{i,j} & a_{i,j+1} & \cdots & a_{i,n-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,j} & a_{n-1,j+1} & \cdots & a_{n-1,n-1} \end{array}\right]$$

telles que :

1. toutes les entrées de $a^{(1)}$ sont plus petites que `x` ;
2. toutes les entrées de $a^{(4)}$ sont plus grandes que `x`.

Si on arrive à procéder à une recherche telle qu'à chaque étape, on peut assurer que toutes les entrées de $a^{(3)}$ sont différentes de `x`, alors on peut en effet se contenter d'explorer $a^{(2)}$, i.e. on peut se contenter de chercher *vers la droite* et *vers le haut*.

```
1  public boolean isInMatrix(Comparable[][] a, Comparable x) {
2    int n = a.length;
3    int i = 0, j = 0;
4    while(i < n-1 && x.compareTo(a[i+1][j]) >= 0)
5      i++;
6    if(x.compareTo(a[i][j]) == 0)
7      return true;
8    while(i >= 0 && j < n-1) {
9      j++;
10     while(i > 0 && x.compareTo(a[i-1][j]) <= 0)
11       i--;
```

2. Boyer, Robert S., and J. Strother Moore. "MJRTY—a fast majority vote algorithm." Automated Reasoning. Springer, Dordrecht, 1991. 105-117.
3. Hesselink, Wim H. "The Boyer-Moore Majority Vote Algorithm." whh348–1-2 (2005)

```
12      if(x.compareTo(a[i][j]) == 0)
13        return true;
14    }
15    return false;
16  }
```

Montrons que l'algorithme retourne `true` si et seulement si `x` est dans `a`.

Si `x` n'est pas dans, il est trivial que l'algorithme ne retournera pas `true` car à aucun moment la comparaison ne pourrait être vraie. Comme le programme se finit, il est clair qu'il renverra `false`.

Si `x` est dans `a`, alors montrons qu'à chaque étape, soit `a[i][j]=x`, soit `x` n'apparaît pas dans $a^{(3)}$. Montrons cela par récurrence :

Cas de base au début de l'algorithme, `j=0` et `i` est le plus grand indice tel que `a[i][j]>x` et `a[i-1][j]<=x`. Si `a[i-1][j]=x`, alors l'algorithme retourne `true` (ligne 7). Sinon la sous-matrice $a^{(3)}$ contient uniquement les éléments de `a[][0]` qui sont plus grands que `x` puisque `a` est triée.

Pas de récurrence Supposons qu'arrivés à la colonne `j`, l'algorithme donne l'indice `i` tel que la sous-matrice $a^{(3)}$ ne contient pas l'élément `x`. Alors à la colonne `j+1`, l'algorithme détermine le nouvel indice `new_i` tel que `new_i<=i`, `a[new_i+1][j+1]>x` et `a[new_i][j+1]<=x`. Si `a[new_i][j+1]=x`, alors l'algorithme renvoie `true` (ligne 13). L'élément `x` ne peut pas se trouver dans $a^{(1)}$ ni dans $a^{(4)}$ par transitivité, et ne peut se trouver dans $a^{(3)}$ non plus car par hypothèse de récurrence `x` ne se trouve pas dans les `j` premières colonnes, et `x` ne peut pas se trouver dans la colonne `j+1` non plus par construction de `new_i`.

Il reste à montrer que l'algorithme effectue bien $O(n)$ comparaisons dans le pire des cas. Intuitivement, chaque colonne est visitée au plus une fois, et chaque ligne également. Puisque le nombre de comparaisons effectuées à chaque indice `(i,j)` est un $O(1)$, le nombre de comparaisons totales est bien $O(n)$

De manière plus précise :

1. la première boucle (ligne 4) effectue au plus n comparaisons ;
2. la condition de la boucle sur i (ligne 10) est évaluée au moins une fois par instance de la boucle générale (ligne 8), i.e. minimum n comparaisons sont effectuées ;
3. à chaque fois que cette condition est vérifiée, une nouvelle comparaison doit être effectuée, et puisque cette condition ne peut être vérifiée que n fois maximum, maximum $2n$ comparaisons sont effectuées à la ligne 10 ;
4. la condition de la ligne 12 est évaluée une fois par instance de la boucle générale (ligne 8), i.e. maximum n comparaisons sont effectuées ici.

En ajoutant la comparaison de la ligne 6 et celle de la ligne 15, on peut en effet finalement borner le nombre de comparaisons par $4n + 2 = O(n)$. □

Séance 2 — Union-Find

Exercice 2.1. Donner une implémentation récursive de la compression de chemin dans la méthode `find`.

Résolution. La méthode `find` doit à la fois compresser le chemin (i.e. s'assurer que la branche allant de la racine à l'élément `p` soit aplatie et que donc tous ses éléments aient la racine pour parent) et retourner un représentant de la classe de `p`. Dès lors si `find` retourne systématiquement la racine de l'arbre, alors il suffit de mettre à jour le parent à chaque appel récursif :

```
1 public int find(int p) {
2   if(parent[p] != p)
3     parent[p] = find(parent[p]);
4   return parent[p];
5 }
```

□

Exercice 2.2. Trouver l'erreur dans cette écriture de la méthode `union` pour l'algorithme naïf :

```
1 public void union(int p, int q) {
2   if (connected(p, q))
3     return;
4   for (int i = 0; i < id.length; i++)
5     if (id[i] == id[p])
6       id[i] = id[q];
7   count--;
8 }
```

Résolution. La valeur `id[p]` n'est pas conservée. Dès lors elle sera mise à jour et remplacée par `id[q]` pendant la boucle `for`. Tout élément de la même classe d'équivalence que `p` mais représenté par un indice plus grand que `p` ne sera donc pas mis à jour car l'égalité ne sera plus testée avec `id[p]` mais bien avec `id[q]`.

La bonne approche est donc de copier la valeur de `id[p]` dans une variable `pID` avant la boucle, et d'effectuer la comparaison entre `id[i]` et `pID` au lieu de `id[i]` et `id[p]`. □

Exercice 2.3. Lesquels de ces tableaux `parent` ne peuvent pas être obtenus par la méthode d'union rapide pondérée avec compression de chemin ?

1. 0 1 2 3 4 5 6 7 8 9
2. 7 3 8 3 4 5 6 8 8 1
3. 6 3 8 0 4 5 6 9 8 1
4. 0 0 0 0 0 0 0 0 0 0
5. 9 6 2 6 1 4 5 8 8 9
6. 9 8 7 6 5 4 3 2 1 0

Résolution. Il est évident que le premier est un tableau valide puisque c'est la valeur initiale du tableau `parent` lorsque toutes les classes d'équivalence ne contiennent qu'un unique élément (Figure 1).

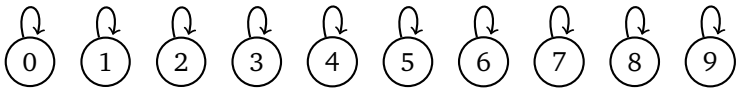
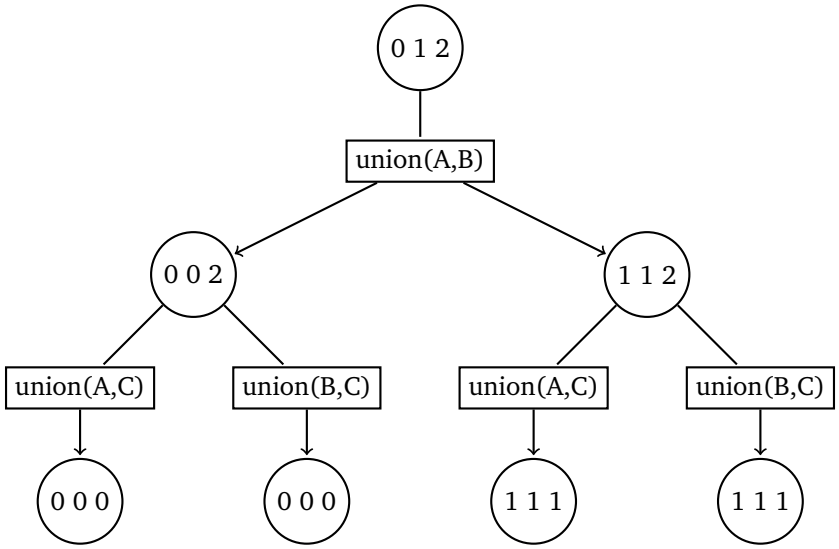


FIGURE 1 – Structure induite par le tableau 1.

Le deuxième tableau ne peut pas être obtenu après union rapide avec compression de chemin car il est impossible pour une classe de trois éléments d'être représentée par une chaîne (Figure 2). En effet, si on a trois éléments `A`, `B` et `C` qui sont initialement disjoints, à permutation des éléments, on peut supposer sans perte de généralité que l'on unit d'abord `A` et `B` ; et ensuite soit `A` et `C`, ou `B` et `C` :



Dans les deux cas, le tableau `parent` contient la même valeur répétée trois fois, donc un des deux éléments est son propre parent (i.e. la racine), et les deux autres sont des enfants directs de cette racine.

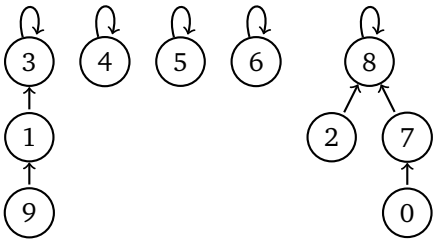


FIGURE 2 – Structure induite par le tableau 2.

Le troisième tableau non plus car il est évident que la branche descendant de l'élément `6` n'est absolument pas balancée (Figure 3).

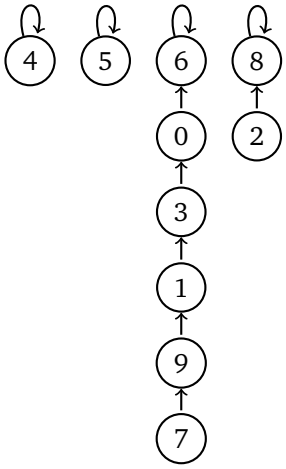


FIGURE 3 – Structure induite par le tableau 3.

Le quatrième quant à lui peut tout à fait être obtenu en ajoutant successivement les éléments `1` à `9` à la classe de `0` (Figure 4).

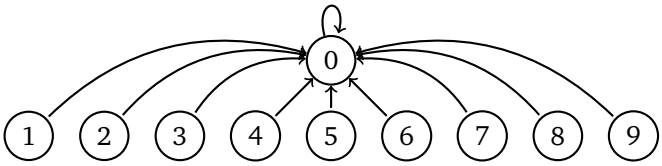


FIGURE 4 – Structure induite par le tableau 4.

Les tableaux 5 et 6 contiennent tous deux des cycles dirigés de longueur > 1, ce qui est impossible par construction de l'*arbre* d'une classe d'équivalence (Figure 5 et Figure 6). De plus le sixième tableau ne contient aucun cycle de longueur 1 (i.e. aucun élément qui est son propre parent), et donc aucune racine, ce qui est à nouveau impossible par construction.

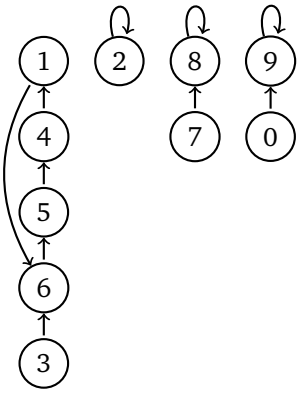


FIGURE 5 – Structure induite par le tableau 5.

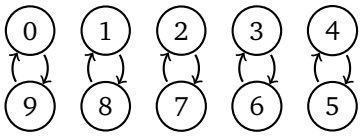


FIGURE 6 – Structure induite par le tableau 6.

□

Exercice 2.4. Étant donnés n éléments, donner une suite de $\sim n$ opérations `union` (avec l'algorithme d'union rapide et la compression de chemin) telle que la hauteur d'au moins un des arbres construits soit $\Theta(\log n)$.

Résolution.

```
UF uf = UF((int)Math.pow(2, N));
for(int j = 0; j < N; j++) {
    for(int k = 0; k < (int)Math.pow(2, N-j-1); k++) {
        int tmp = (int)Math.pow(2, j);
        uf.union(2*k * tmp, (2*k + 1) * tmp);
    }
}
```

pour $N = 5$, on peut voir que l'arbre engendré par cette construction a en effet une hauteur de $5 = \log_2 32 = \log_2 2^N \sim \log_2(2^N + 1)$. Il est en effet assez clair que si le nombre d'éléments dans la structure est une puissance de 2 ($n = 2^N$), alors à la fin de l'étape `j`, le nombre d'arbres dans la forêt est 2^{N-j-1} (i.e. on divise le nombre de classe d'équivalences par 2 à chaque étape puisqu'on unit 2^{N-j-1} classes distinctes . De plus, il est clair que tous ces arbres ont la même hauteur, et que cette hauteur est augmentée de 1 à chaque étape puisque la racine de l'un est mise comme enfant de la racine de l'autre[4]). La hauteur de l'arbre à la fin du procédé est donc égal au nombre d'étapes, or il y a $\log_2 n$ étapes.

Montrons maintenant qu'il y a bien $\sim n$ opérations `union`. Puisque une étape j ($0 \leq j \leq N-1$) consiste en 2^{N-j-1} opérations `union`, le nombre total d'opérations `union` est $1 + 2 + ... + 2^{N-1} = 2^N - 1 \sim n$.

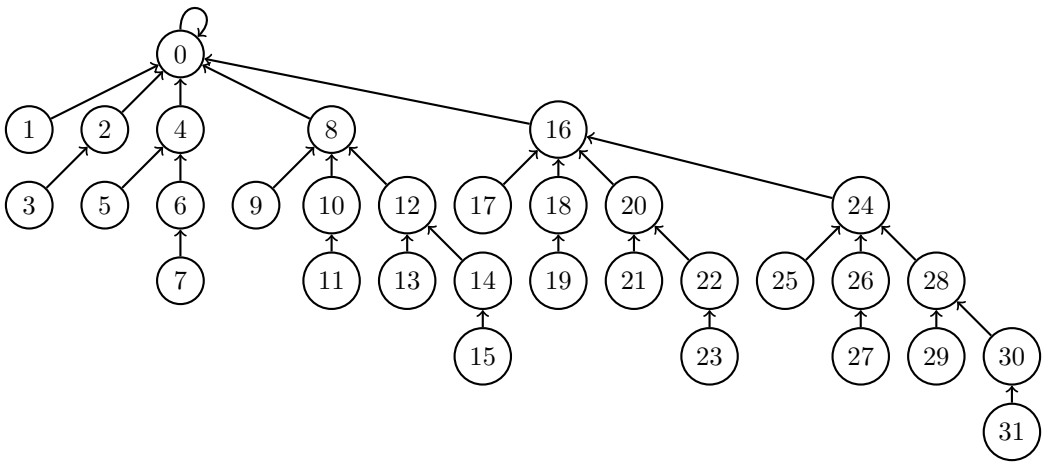


FIGURE 7 – Structure interne d'une instance de union-find ayant un unique arbre de hauteur $\sim \log_2 n$.

4. Puisqu'on fait l'union de deux racines, il n'y a pas compression de chemin.

Les étapes intermédiaires de cette construction sont les suivantes :

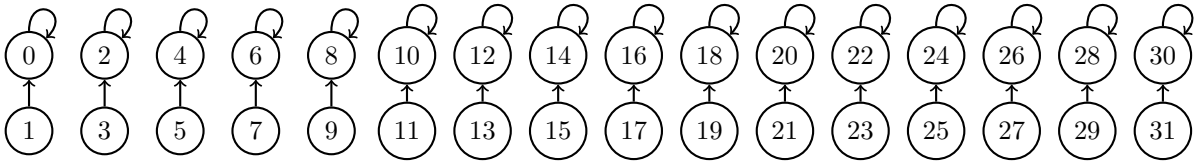


FIGURE 8 – Étape 1 de la construction de la solution de l'exercice 2.4.

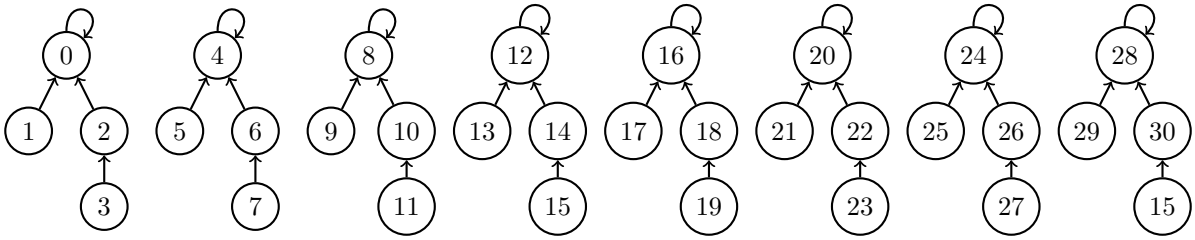


FIGURE 9 – Étape 2 de la construction de la solution de l'exercice 2.4.

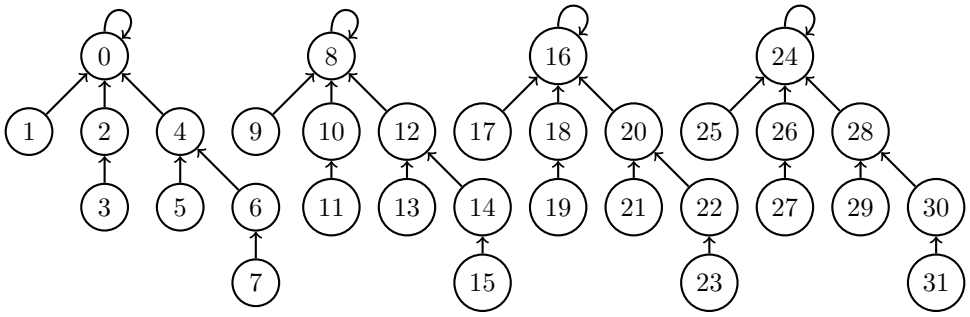


FIGURE 10 – Étape 3 de la construction de la solution de l'exercice 2.4.

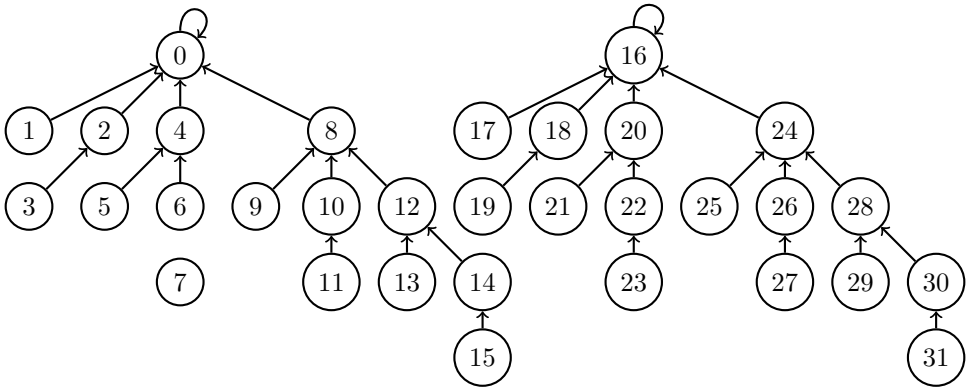


FIGURE 11 – Étape 4 de la construction de la solution de l'exercice 2.4.

Dans le cas où n n'est pas une puissance de 2, on peut écrire le code suivant, plus général mais dont le résultat est équivalent :

```
1  UF uf = UF(n);
2  int m = 1;
3  int k = 0;
4  while(m < n) {
5      k = 0;
6      while((2*k+1)*m < n) {
7          uf.union(2*k * m, (2*k+1) * m);
8          k++;
9      }
10     m *= 2;
11 }
```

Le nombre d'appels à `union` effectués est donc de $\lfloor \frac{n}{2} \rfloor$ à la première étape, de $\lfloor \frac{n}{4} \rfloor$ à la seconde étape, etc. jusqu'à un unique appel à la dernière étape. La première boucle `while` (ligne 4) est exécutée $\lfloor \log_2 n \rfloor$ fois, et la seconde est exécutée $\lfloor \frac{n}{m} \rfloor$ fois. Le nombre total d'appels à `union` est donc :

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \ldots + 1 = \sum_{j=1}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^j} \right\rfloor = n - 1.$$

En effet, à la fin de l'étape `j`, tous les éléments de la forme $(2k+1)2^j$ ont été liés, et ne seront plus modifiés. En incrémentant j au fur et à mesure, toutes les valeurs seront donc considérées exactement une fois jusqu'à ce qu'il ne reste plus qu'un unique élément non-lié (l'élément 0, la racine). Le nombre d'appels à `union` est donc $n-1$. De plus, la hauteur de l'arbre est une fonction croissante du nombre d'éléments présents dans la structure. Donc :

$$\forall N > 0, p < 2^N : h(2^N) \leq h(2^N + p) \leq h(2^{N+1}),$$

i.e. :

$$\forall N > 0, p < 2^N : N \leq h(2^N + p) \leq N + 1.$$

La hauteur de l'arbre est donc bien $\Theta(\log_2 n)$ (et même $\sim \log_2 n$). □

Exercice 2.5. Dans la méthode d'union rapide, on attache toujours l'arbre de hauteur plus petite (s'il y en a un) à celui de hauteur plus grande. Que se passe-t-il si on remplace dans cette méthode la hauteur par le nombre d'éléments ?

Résolution. Dans ce cas, on a également que la hauteur d'un arbre dans la forêt construite par la procédure d'union rapide pondérée pour n éléments est au plus $\log n$. Prouvons par induction qu'un arbre de hauteur h, construit de cette manière, contient toujours au moins 2^h éléments.

La proposition est vraie à l'initialisation de la structure. Montrons que la proposition reste vraie après une union. Par hypothèse d'induction, les deux arbres de `p` et `q` de hauteur respective h_1 et h_2 contiennent $n_1 \geq 2^{h_1}$ et $n_2 \geq 2^{h_2}$ éléments. On suppose sans perte de généralité que $n_1 \leq n_2$ et qu'on attache donc l'arbre de `p` à la racine de celui contenant `q`. Si $h_1 < h_2$, la hauteur du nouvel arbre est h_2 et contient $n = n_1 + n_2 > n_2 \geq 2^{h_2}$ éléments donc la proposition reste vraie pour le nouvel arbre. Si, par contre, $h_1 \geq h_2$, la hauteur du nouvel arbre est $h_1 + 1$. Cet arbre contiendra $n = n_1 + n_2 \geq 2n_1 \geq 2^{h_1+1}$ éléments et la proposition reste également vraie pour le nouvel arbre. Enfin si un arbre de hauteur h contient $n \geq 2^h$ éléments, on a bien $h \leq \log_2 n$. □

Séance 3 — Tris

Exercice 3.1 (Le drapeau tricolore)**.** Décrire un algorithme qui, en une passe dans un tableau de taille n avec v comme premier élément, partitionne les éléments en trois groupes successifs : les éléments strictement inférieurs à v, les éléments égaux à v, et ceux strictement supérieurs à v. On peut utiliser un opérateur de comparaison ternaire.

Résolution. Etant donné v, on souhaite partitionner le tableau tel qu'il existe des indices l et h tel que toutes les valeurs inférieures à v se trouvent avant l et toutes les valeurs supérieures à v se trouvent après $h-1$.

On initialise `i=1`, `l=0` et `h=n`, et, tant que `i` est inférieur à `h` :

1. si `a[i]<v` : swap `a[l]` et `a[i]` incrémente `l` et `i` (on a donc ajouté un élément à la partition inférieure)
2. si `a[i]>v` : décrémente `h` et swap `a[h]` et `a[i]` (on a ajouté un élément à la partition supérieure)
3. sinon : incrémente `i` (on a ajouté un élément à la partition égale)

Notons que quand `a[i]>v` on n'incrémente pas `i`. En effet, suite à l'échange, l'élément en `a[i]` doit à nouveau être évalué.

Comme l'illustre la Figure 12, on a à tout moment : (i) entre 0 et $l-1$ les éléments rencontrés inférieurs, (ii) entre l et $i-1$ les éléments rencontrés égaux à v, (iii) entre i et $h-1$ les éléments restants à parcourir, et (iv) entre h et $n-1$ les éléments rencontrés supérieurs à v.

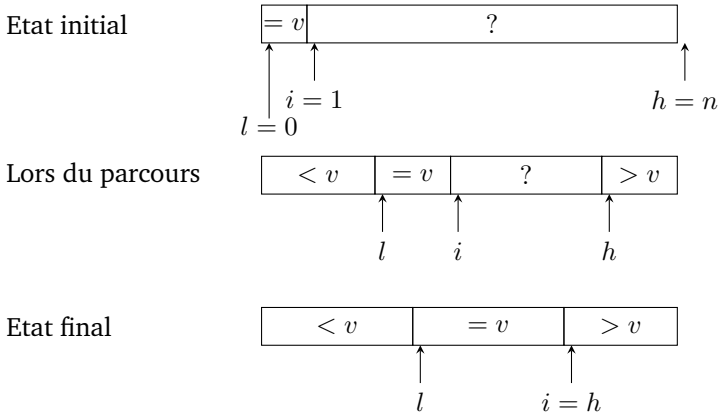


FIGURE 12 – Etat du tableau.

```
1  public void partition(int[] a) {
2    int n=a.length, l=0, h=n, v=a[0];
3
4    int i=1;
5    while (i<h){
6      if (a[i]<v){
7          swap(a,i++,l++);
8      } else if (a[i]>v){
9          swap(a,--h,i);
10     } else {
11         i++;
12     }
13   }
14 }
```

A chaque itération on incrémente `i` ou on décrémente `h`, il suit que le nombre d'itérations est n.

□

Exercice 3.2 (Boulons et écrous)**.** On se donne un ensemble de n écrous et n boulons. Chaque écrou peut être assemblé avec exactement un boulon, et chaque boulon ne convient qu'à un seul écrou. On peut comparer un écrou et un boulon, et déterminer lequel des deux est le plus grand. Mais on ne peut comparer deux boulons ou deux écrous. Donner un algorithme randomisé qui effectue $O(n \log n)$ comparaisons boulon-écrou en moyenne, et qui associe chaque écrou au bon boulon.

Résolution. Étant donné un boulon b choisi au hasard, on peut le comparer à tous les écrous et ainsi obtenir une partition des écrous en trois parties en $O(n)$ opérations (cf. exercice précédent). La partition centrale contiendra donc l'écrou e qui correspond au boulon b. En utilisant cet écrou comme pivot, on peut alors partitionner l'ensemble des boulons. En répétant cette séquence d'actions de façon dichotomique sur chaque moitié on obtient au final deux ensembles triés.

On effectue donc à chaque étape $O(n)$ opérations afin de partitionner les différents sous-ensembles. Et on répète ces étapes $O(\log n)$ fois en moyenne. On peut également adapter la preuve de la complexité de `quicksort`, mais comme nous faisons deux appels à `partition`, nous avons :

$$C(n) = 2n + 1 + \frac{2}{n} \sum_{k=0}^{n-1} C(k),$$

qui mène à l'égalité suivante :

$$nC(n) - (n-1)C(n-1) = 4n + 2C(n-1).$$

La conclusion est donc, par télescopage : $C(n) \sim 4n \log n$.

Il s'agit essentiellement d'un Quicksort modifié. La différence principale étant qu'on doit partitionner en deux étapes, d'abord les boulons et ensuite les écrous. La complexité $O(n \log n)$ est cependant la même.

```
1  public void associer(int[] boulons, int[] ecrous, int lo, int hi)
2  {
3     if (lo < hi) {
4        // sélection d'un pivot aléatoire
5        int boulon_pivot = ThreadLocalRandom.current().nextInt(lo, hi+1);
6
7        // partition des écrous avec le boulon sélectionne aléatoirement
8        int idx_pivot = partition(ecrous, lo, hi, boulons[boulon_pivot]);
9
10       // partition des boulons avec l'écrou pivot
11       partition(boulons, lo, hi, ecrous[idx_pivot]);
12
13       // Appels récursifs sur la partie inférieure...
14       associer(boulons, ecrous, lo, idx_pivot-1);
15       // et supérieure.
16       associer(boulons, ecrous, idx_pivot+1, hi);
17    }
18 }
19
20 public int partition(int[] a, int lo, int hi, int pivot) {
21    int i = lo, l = lo;
```

```
22      while(i<hi) {
23
24         if (a[i]<pivot) {
25               swap(a,i++,l++);
26         } else if (a[i]==pivot){
27            // on préserve temporairement l'élément égal au pivot en fin de tableau
28            swap(a,i,hi);
29         } else {
30            i++;
31         }
32      }
33      // en sortant de la boucle:
34      //    les éléments de lo a l-1 sont inférieurs
35      //    les éléments de l a hi-1 sont supérieurs
36      //    l'élément en hi est egal au pivot
37      // on replace le pivot au centre
38      swap(a,l,hi);
39
40      // retourne l'indice de l'élément pivot
41      return l;
42   }
```

□

Exercice 3.3. Supposons que nous avons un tableau `a[]` de n éléments tel que chaque élément est à distance au plus k de sa position dans le tableau trié. Donner un algorithme qui permet de trier un tel tableau en $O(n \log k)$ comparaisons au pire cas.

Résolution. Posons a_i l'élément qui prendra la position i après le tri. Clairement, chaque a_i se trouvera entre les indices $i-k$ et $i+k$ avant le tri. Il suit donc que les k premiers éléments après tri du tableau se trouvent entre les indices 0 et $2k-1$. En effectuant un tri de ces $2k$ premières position, on s'assure que les k premiers éléments du tableau sont à leur position finale.

Ceux qui se trouvent en position triée finale entre k et $2k-1$ se trouvaient dans le tableau de départ entre 0 et $3k-1$. Dès lors, après le premier tri, ils se trouvent entre k et $3k-1$. Il suffit donc de faire un deuxième tri entre k et $3k-1$ pour les mettre à la bonne place. On peut donc à chaque itération trier une fenêtre de $2k$ éléments par pas de k.

Ainsi on effectue au plus $\lceil n/k \rceil$ tris de $2k$ éléments. La complexité de chacun de ces tris est $O(2k \log(2k)) = O(k \log k)$, on a donc au final $O(\lceil n/k \rceil k \log k) = O(n \log k)$ opérations.

```
1  import java.util.Arrays;
2  public void kSort(int[] a, int k) {
3      int n = a.length;
4      for(int i=0;i<Math.ceil(n/(double)k);i++) {
5          Arrays.sort(a, i*k, Math.min(n,(i+2)*k));
6      }
7  }
```

□

Exercice 3.4. Décrire un algorithme qui étant donnés trois tableaux triés `a[]`, `b[]` et `c[]`, chacun de taille n, les fusionne dans un seul tableau trié `d[]` en effectuant au plus $\sim 6n$ comparaisons. Peut-on réduire le nombre de comparaisons à $\sim 5n$?

Résolution. Si on adapte le code de fusion de deux tableaux triés pour incorporer un troisième tableau, on obtient le code suivant :

```
public int[] merge(int[] a, int[] b, int[] c) {
  int n = a.length;
  int[] d = new int[3*n];
  int i = 0, j = 0, k = 0;
  for(int l = 0; l < 3*n; ++l) {
    if(i < n && (j >= n || a[i] < b[j])) {
      if(k >= n || a[i] < c[k])
        d[l] = a[i++];
      else
        d[l] = c[k++];
    } else {
      if(j < n && (k >= n || b[j] < c[k]))
        d[l] = b[j++];
      else
        d[l] = c[k++];
    }
  }
  return d;
}
```

Le nombre de comparaisons dans le pire des cas est $6n$ puisque la boucle `for` s'exécute $3n$ fois, et puisque à chaque tour de boucle on compare `a[i]` et `b[j]`, et ensuite (selon le résultats de cette comparaison), on compare soit `a[i]` et `c[k]`, soit `b[j]` et `c[k]`. On fait donc deux comparaisons par tour de boucle, i.e. $2 \times 3n = 6n$ comparaisons dans le pire des cas. Notons que le reste des conditions permet de s'assurer que l'on n'accède pas à un élément hors d'un tableau.

On peut cependant faire mieux que $6n$ comparaisons. En effet, la fusion de deux tableaux triés de taille n se fait en $\sim 2n$ comparaisons. De manière plus générale, la fusion de deux tableaux triés `a` et `b` de taille respective m et n se fait en $\sim m+n$ comparaisons. Dès lors, si on fusionne d'abord `a` et `b` en le stockant dans un tableau `tmp`, on effectue au pire $n+n = 2n$ comparaisons. Si on fusionne ensuite `tmp` et `c`, on effectue $2n+n = 3n$ comparaisons. Le nombre total est donc $2n+3n = 5n$ dans le pire des cas.

```
private int[] mergeTwoArrays(int[] a, int[] b) {
  int n1 = a.length;
  int n2 = b.length;
  int[] d = new int[n1+n2];
  int i = 0, j = 0, l = 0;
  while(i < n1 && j < n2) {
    if(a[i] < b[j])
      d[l++] = a[i++];
    else
      d[l++] = b[j++];
  }
  while(i < n1)
    d[l++] = a[i++];
```

```
14    while(j < n2)
15      d[l++] = b[j++];
16    return d;
17  }
18
19  public int[] merge(int[] a, int[] b, int[] c) {
20    int[] tmp = mergeTwoArrays(a, b);
21    return mergeTwoArrays(tmp, c);
22  }
```

bien que cette approche nécessite moins de comparaisons dans le pire des cas que le précédent, elle requiert plus de mémoire temporaire. En effet, en plus des trois tableaux de taille n et du tableau de taille $3n$ dans lequel le résultat est stocké, on a besoin d'un tableau de taille $2n$ pour stocker la fusion de `a` et `b`. □

Exercice 3.5. Démontrer que le problème de fusion de trois tableaux de l'exercice précédent ne peut être résolu en moins de $4.754n$ comparaisons au pire cas.

Résolution. **Remarque : correction en cours de révision.**

Remarquons que $4.754 \simeq 3 \log_2 3$. Montrons ce résultat de manière plus générale. Montrons que le problème de fusion de k tableaux triés de taille n ne peut être résolu en moins de $nk \log_2 k$ comparaisons au pire des cas.

Supposons par l'absurde que l'on a un algorithme qui est capable de fusionner k tableaux triés de taille n en $c(k, n) \lneqq nk \log_2 k$ opérations dans le pire des cas. Prenons alors un tableau `x` quelconque de taille kn. On peut alors diviser ce tableau en k tableaux de taille n :

x	a_1	a_2	$\cdots$	a_{k-1}	a_k

Chacun de ces tableaux a_i peut alors être trié par merge sort pour un total de $k \times n \log_2 n$ comparaisons dans le pire des cas. Il nous reste alors à fusionner ces k tableaux, pour un total dont le nombre de comparaisons dans le pire des cas est :

$$nk \log_2 n + c(k, n) \lneqq nk \log_2 n + nk \log_2 k = nk \left(\log_2 n + \log_2 k\right) = nk \log_2(nk).$$

Or vous avez vu au cours (Proposition 8) que tout algorithme de tri d'un tableau de taille fixée N fera toujours au moins $N \log_2 N$ comparaisons dans le pire des cas. Nous avons donc une contradiction, et un tel algorithme ne peut exister.

Dans le cas $k = 3$ (le cas de l'exercice précédent), on voit bien que le nombre minimum de comparaisons à effectuer si les tableaux `a`, `b` et `c` sont de taille n, est $nk \log_2 k = n3 \log_2 3 \simeq 4.754n$. □

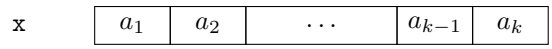
Résolution. **Remarque : correction alternative.**

Remarquons que $4.754 \simeq 3 \log_2 3$. Montrons ce résultat de manière plus générale. Montrons que le problème de fusion de k tableaux triés de taille n ne peut être résolu en moins de $nk \log_2 k - o(n)$ comparaisons au pire des cas.

Soit $0 < b < 1$ une constante. Supposons par l'absurde que l'on a un algorithme qui est capable de fusionner k tableaux triés de taille n en au plus $nk \log_2(k \cdot b)$ comparaisons dans le pire des cas. Nous allons montrer par récurrence su i qu'avec cette supposition, on peut pour tout entier $i \geq 0$ obtenir un algorithme qui trie tout tableau de taille n en au plus $n \log_2(n \cdot b^i) + o(n)$ comparaisons dans le pire des cas.

Vérifions tout d'abord le cas de base (initialisation) $i = 0$. Nous savons que l'algorithme merge sort effectue au plus $n \log_2 n + o(n)$ comparaisons dans le pire des cas. Donc ce cas est bien vérifié.

Passons maintenant à l'hérédité. Soit $i \geq 0$ un entier, et supposons que la propriété pour i est établie, c'est à dire qu'il existe un algorithme qui trie tout tableau de taille n en au plus $n \log_2(n \cdot b^i) + o(n)$ comparaisons dans le pire des cas. Montrons que cette propriété reste vraie pour $i+1$. Pour cela, considérons un tableau `x` quelconque de taille $k \cdot n$. On peut alors diviser ce tableau en k tableaux, chacune de taille n :



Par hypothèse de récurrence, chacun de ces tableaux peut être trié pour un total de $k \times n \log_2(n \cdot b^i) + o(n)$ comparaisons dans le pire des cas. Il nous reste alors à fusionner ces k tableaux, pour un total dont le nombre de comparaisons dans le pire des cas est :

$$nk \log_2(n \cdot b^i) + nk \log_2(k \cdot b) + o(nk) = nk \log_2(nk \cdot b^{i+1}) + o(nk).$$

On obtient donc un algorithme de tri qui effectue au plus $n \log_2(n \cdot b^{i+1}) + o(n)$ comparaisons dans le pire des cas pour un tableau de taille n, et l'hérédité est démontrée. Par récurrence, la proposition est vraie pour tout entier $i \geq 0$.

Pouvons-nous maintenant obtenir une contradiction ? Notons tout d'abord que b étant compris strictement entre 0 et 1, il existe i tel que $n \cdot b^i < \frac{n}{2e}$, et nous avons donc démontré l'existence d'un algorithme triant un tableau quelconque de taille n en au plus $n \log_2 \frac{n}{2e} + o(n)$ comparaisons. Nous avons vu dans le cours (Proposition 8) que tout algorithme de tri doit faire au moins $\log_2(n!)$ comparaisons dans le pire des cas. La formule de Stirling (Rappel Mathématique 5) permet d'obtenir $\log_2(n!) = n \log_2 \frac{n}{e} + o(n)$. L'algorithme que nous avons trouvé effectue donc moins de comparaisons que le minimum possible, et nous avons une contradiction.

Nous en concluons donc que pour tout $0 < b < 1$, il n'existe pas de d'algorithme qui est capable de fusionner k tableaux triés de taille n en au plus $nk \log_2(k \cdot b)$ comparaisons dans le pire des cas. Il en découle qu'un tel algorithme doit effectuer au moins $\sim nk \log_2(k)$ comparaisons dans le pire des cas.

Dans le cas $k = 3$ (le cas de l'exercice précédent), on voit bien que le nombre minimum de comparaisons à effectuer si les tableaux `a`, `b` et `c` sont de taille n, est $\sim nk \log_2 k \sim n3 \log_2 3 \simeq 4.754n$. □

Exercice 3.6. Étant donnés deux tableaux triés `a[]` et `b[]` de tailles respectives m et n, où $m \geq n$, donner un algorithme qui les fusionne dans un nouveau tableau `c[]` en utilisant $\sim n \log_2 m$ comparaisons au pire cas.

Résolution. Puisque `a` est plus grand que `b`, on s'attend à avoir de *longues* chaînes d'éléments de `a` qui vont être ajoutées entre chaque ajout d'éléments de `b`. Au lieu de systématiquement comparer l'élément de `b` actuel avec tous les éléments de `a` jusqu'à en trouver un plus grand, on peut appliquer une recherche dichotomique au sein des éléments de `a`.

```
1 // trouve l'index du premier élément > x dans le tableau et retourne a.length si tous
      les éléments sont <= x;
2 private int search(int[] a, int x, int lo, int hi) {
3   while(lo < hi) {
4     int mid = (lo+hi) / 2;
5     if(a[mid] <= x)
6       lo = mid+1;
7     else
8       hi = mid;
```

```
  }
  return lo;
}

public int[] merge(int[] a, int[] b) {
  int m = a.size;
  int n = b.size;
  if(n > m)
    return merge(b, a);
  int[] c = new int[m+n];
  int lo = 0, hi = m, l = 0;
  for(int j = 0; j < n; ++j) {
    int idx = search(a, b[j], lo, hi);
    while(lo < idx)
      c[l++] = a[lo++];
    c[l++] = b[j];
  }
  while(lo < hi)
    c[l++] = a[lo++];
  return c;
}
```

□

Séance 4 — Tri par tas

Exercice 4.1. Supposons qu'une application requiert un grand nombre d'insertions mais seulement très peu de suppressions du maximum. Laquelle de ces implémentations serait la plus efficace : un tas, un tableau trié, un tableau non trié ?

Résolution. Pour un tas, une insertion ainsi que la suppression du maximum prennent un temps $O(\log n)$. Pour un tableau trié, une insertion prend un temps $O(n)$ (trouver l'indice où insérer l'élément prend un temps $O(\log n)$ tandis que déplacer d'une case tous les éléments plus grands prend un temps $O(n)$) alors que la suppression du maximum prend un temps constant. Finalement, pour un tableau non trié, une insertion prend un temps constant tandis que la suppression du maximum prend un temps $O(n)$. On voit que, dans ce cas, il vaut mieux utiliser un tableau non trié. □

Exercice 4.2. Supposons qu'une application requiert un grand nombre de consultations du maximum mais seulement très peu de suppressions du maximum et d'insertions. Laquelle de ces implémentations serait la plus efficace : un tas, un tableau trié, un tableau non trié ?

Résolution. Pour un tas ainsi que pour un tableau trié, la consultation du maximum prend un temps constant. Pour un tableau non trié, elle prend un temps $O(n)$. Le tableau trié étant plus simple qu'un tas dans son utilisation, il est recommandé d'utiliser une telle structure. □

Exercice 4.3. Donner un algorithme en temps linéaire qui vérifie si un tableau donné en entrée représente un tas.

Résolution. On procède en parcourant le tableau et en vérifiant, pour chaque élément, qu'il est bien plus grand que ses deux "enfants".

```
1  public boolean isHeap(int[] a) {
2          int n = a.length;
3          for (int i = 0; i < n; i++) {
4              int parent = a[i];
5              if (2*i+1 < n) {
6                  int child1 = a[2*i+1];
7                  if (parent < child1) return false;
8              }
9              if (2*i+2 < n) {
10                 int child2 = a[2*i+2];
11                 if (parent < child2) return false;
12             }
13         }
14         return true;
15     }
```

□

Exercice 4.4 (Maintenir la médiane)**.** On définit la médiane d'un ensemble de n nombres distincts comme le nombre de l'ensemble qui a exactement $\lfloor (n-1)/2 \rfloor$ éléments plus petits, et $\lceil (n-1)/2 \rceil$ éléments plus grands. Décrire une structure de données qui contient un ensemble de nombres et permet les opérations suivantes : insertion en temps logarithmique, trouver la médiane en temps constant, supprimer la médiane en temps logarithmique.

Résolution. Utilisons pour cela deux tas, un tas-MAX et un tas-MIN. Dans le tas-MAX, chaque élément est plus grand que ses enfants, tandis que dans le tas-MIN, chaque élément est plus petit que ses enfants. La médiane de l'ensemble se trouve à la racine des deux tas. Considérons par exemple l'ensemble représenté à la Figure 13.

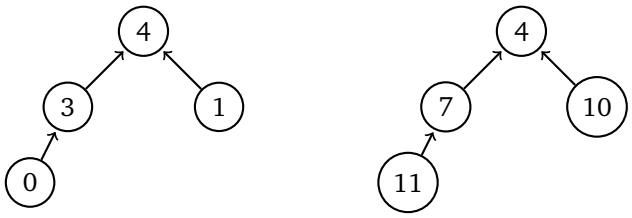


FIGURE 13 – Exemple de tas-MAX et tas-MIN pour la maintenance de la médiane.

L'élément 4, à la racine des deux tas, est bien la médiane de l'ensemble qui comporte 7 éléments. Il y a en effet trois éléments plus petits et trois éléments plus grands que 4 et les deux tas sont bien équilibrés. Supposons que l'on rajoute l'élément 8 à l'ensemble. Puisqu'il est plus grand que 4, nous allons le mettre dans le tas-MIN, à droite (voir Figure 14).

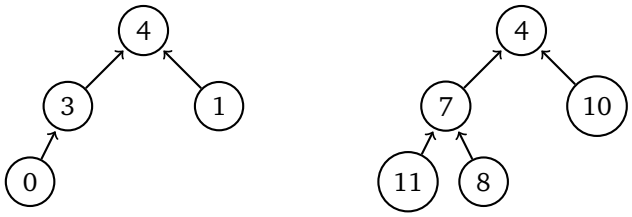


FIGURE 14 – Exemple de tas-MAX et tas-MIN pour la maintenance de la médiane.

Notons que la racine du tas-MIN est inchangée. Il faut toutefois se demander ici si 4 reste la médiane de l'ensemble. Puisqu'il y a 8 éléments, la médiane doit avoir 3 éléments plus petits et 4 éléments plus grands. Nous voyons que c'est bien le cas ici. Supposons qu'un nouvel élément 12 soit rajouté à l'ensemble. Plaçons le à nouveau dans le tas-MIN (voir Figure 15). Nous voyons que les deux tas sont déséquilibrés et que 4 n'est plus la médiane de l'ensemble. En supprimant la racine du tas-MIN et en rajoutant la nouvelle racine du tas-MIN (7) au tas-MAX, nous obtenons la structure plus équilibrée de la Figure 16.

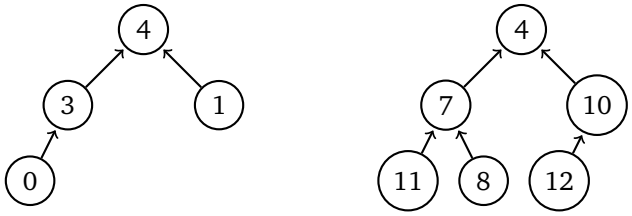


FIGURE 15 – Exemple de tas-MAX et tas-MIN pour la maintenance de la médiane.

De manière générale, à chaque insertion d'un élément, on compare celui-ci à la racine des tas et on le place dans le tas approprié.

Ensuite, si `taille(tas-MAX) > taille(tas-MIN)` ou si `taille(tas-MIN) > taille(tas-MAX)+1`, i.e. si la racine ne correspond plus à la médiane, on supprime la racine du tas le plus encombré et on rajoute sa nouvelle racine à l'autre tas. Comme chacune de ces opérations, l'insertion d'un élément prend donc un temps logarithmique.

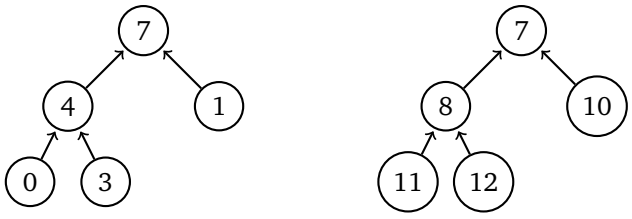


FIGURE 16 – Exemple de tas-MAX et tas-MIN pour la maintenance de la médiane.

Pour ce qui est de la suppression de la médiane, il faut retirer la racine de chacun des deux tas, et s'assurer que la nouvelle médiane y apparaisse bien. La suppression de la racine dans un heap se fait en $O(\log n)$ donc la suppression des deux racines est également en $O(\log n)$. Il faut ensuite distinguer deux cas :

1. `taille(tas-MIN) == taille(tas-MAX)` (i.e. si l'ensemble de nombres considéré est de taille paire) ;
2. `taille(tas-MIN) == taille(tas-MAX)+1` (i.e. si l'ensemble de nombres considéré est de taille impaire).

Il est en effet impossible par construction que `taille(tas-MAX) < taille(tas-MIN)` ou que `taille(tas-MIN) < taille(tas-MAX)-1` sinon la racine des tas ne correspondrait pas à la médiane de l'ensemble.

Dans le premier cas, la nouvelle médiane est la racine du tas-MAX. En effet, notons m la taille des deux tas. Nous avons donc $2m$ éléments au total dans l'ensemble, et tous les éléments du tax-MAX sont $\leqslant$ aux éléments du tas-MIN. Or $\left\lfloor \frac{2m-1}{2} \right\rfloor = n-1$ et $\left\lceil \frac{2m-1}{2} \right\rceil = n$. La racine du tas-MAX satisfait donc bien la définition de la médiane. Il nous faut alors ajouter cette valeur dans le tas-MIN.

Dans le second cas, la nouvelle médiane est la racine du tas-MIN. En effet, notons m la taille du tas-MAX. Nous avons au total $2m+1$ dans l'ensemble et de plus nous savons que $\left\lfloor \frac{2m+1-1}{2} \right\rfloor = m = \left\lceil \frac{2m+1-1}{2} \right\rceil$. Dès lors la racine du tas-MIN satisfait bien la définition de médiane. Il nous faut alors ajouter cette valeur dans le tas-MAX.

Dans les deux cas possibles, après la suppression des deux racines, il reste une unique insertion dans un tas, qui est également logarithmique en temps.

□

Exercice 4.5. Donner un algorithme pour l'insertion dans un tas qui n'effectue que $O(\log \log n)$ *comparaisons* au pire cas.

Résolution. Le principe consiste à insérer l'élément (x) à la première position vide et, ensuite, à trouver, de façon dichotomique, le dernier ancêtre plus petit ou égal à x. C'est à cet endroit que devra se positionner finalement l'élément x. L'ancêtre en question ainsi que ses descendants qui étaient précédemment ancêtres de x seront déplacés d'une unité vers le bas. La recherche du dernier ancêtre plus petit ou égal à x effectue $O(\log \log n)$ comparaisons au pire des cas, puisque la hauteur de l'arbre est $O(\log n)$ et qu'il s'agit d'une recherche dichotomique. Le déplacement des descendants de cet ancêtre vers le bas ne nécessite aucune comparaison.

```
1  class HeapQuickInsert {
2
3      public void insert(int[] a, int size, int x) {
4          a[size] = x; // mettre x a prochaine place libre
5          size++;
6          // vérifier les ancêtres du nouvel élément jusqu'à  ce qu'on trouve le dernier
7          // ancêtre <= x
```

```
        int niveau = (int) Math.floor(Math.log(size)/Math.log(2));
        // n.b. niveau de la racine = 0
        int lo = 0, hi = niveau;
        while (lo < hi) {
            int mid = (lo + hi) / 2;
            int ancetre = size / (int) Math.pow(2, niveau - mid) - 1;
            if (x < a[ancetre])
                lo = mid + 1;
            else
                hi = mid;
        }
        if (lo == niveau) return; // x se trouve déjà  a la bonne place
        // le dernier ancêtre <= x se trouve au niveau lo = hi; il faut mettre x a cet
        // endroit et déplacer les descendants de cet ancêtre qui étaient auparavant
        // ancêtres de x d'une unité vers le bas
        int index_x = size / (int) Math.pow(2, niveau-lo) - 1; // index ou mettre x
        int index = size-1; // index actuel de x
        int index_parent;
        while (index > index_x) {
            index_parent = (index+1)/2-1;
            a[index] = a[index_parent];
            index = index_parent;
        }
        a[index_x] = x;
    }

    public static void main(String[] argv) {
        HeapQuickInsert h = new HeapQuickInsert();
        int[] a = new int[100]; // taille maximale du tas: 100
        a[0]=11; a[1]=9; a[2]=8; a[3]=7; a[4]=6; a[5]=4; // a = {11,9,8,7,6,4}
        int size = 6; // taille réelle du tas
        h.insert(a, size, 10); // a = {11,9,10,7,6,4,8}
        size++;
    }
}
```

□

Exercice 4.6. Prouver qu'il est impossible d'avoir une structure de données pour laquelle tant l'insertion que l'effacement du minimum n'utiliserait qu'au plus $O(\log \log n)$ comparaisons.

Résolution. Dans ce cas, on pourrait trier n éléments en $O(n \log \log n)$ comparaisons : tout d'abord, en insérant les n éléments dans le tas, ce qui nécessiterait $O(n \log \log n)$ comparaisons, et ensuite, en retirant n fois le minimum, ce qui nécessiterait également $O(n \log \log n)$ comparaisons. Or, il a été montré que trier n éléments nécessite au moins $\sim n \log n$ comparaisons. □

Séance 5 — Arbres de recherche

Exercice 5.1. Dessiner l'arbre rouge-noir résultant de l'insertion des clés de A à K dans l'ordre alphabétique. Décrivez de manière générale ce qu'il se produit lorsque des clés sont insérées en ordre croissant dans un arbre rouge-noir initialement vide.

Résolution. L'ajout par étape est illustré par les figures 17 à 20. L'insertion de clés dans l'ordre trié dans un arbre binaire de recherche non-équilibré produit une chaîne. Au contraire, lorsqu'on insère des clés dans l'ordre croissant dans un arbre rouge-noir une rotation rééquilibre l'arbre toutes les deux insertions. Ces rotations assurent la hauteur logarithmique de l'arbre (on remarque que la hauteur de l'arbre est incrémentée uniquement après la 1ère, 2ème, 4ème et la 8ème insertion). □

Exercice 5.2. Donner une méthode `is23()` qui vérifie qu'aucun nœud n'est connecté à deux arêtes rouges successives et qu'il n'y a pas d'arête rouge vers la droite. Donner une méthode `isBalanced()` qui vérifie que tous les chemins de la racine à une feuille ont le même nombre d'arêtes noires. Combiner ces méthodes en une méthode `isRedBlackBST()` qui vérifie que l'arbre est un arbre binaire de recherche et qu'il satisfait ces deux conditions.

Résolution.

```
1  public boolean is23(Node h) {
2      if (h == null) return true;
3      if (isRed(h.right)) return false; // Arête rouge à droite
4      // Deux arêtes rouges sur le chemin gauche
5      if (isRed(h.left) && h.left.left != null && isRed(h.left.left)) return false;
6      return (is23(h.left) && is23(h.right));
7  }
8
9  public int balancedBlackHeight(Node h) {
10     // Renvoie -1 si l'arbre enraciné en h n'est pas équilibré, sinon renvoie sa hauteur
           noire.
11     if (h==null) return 0;
12     int left_height = balancedBlackHeight(h.left);
13     int right_height = balancedBlackHeight(h.right);
14     if (left_height == -1 || left_height != right_height) return -1;
15     return (isRed(h) ? 0 : 1) + left_height;
16 }
17
18 public boolean isBalanced(Node h) {
19     if (h == null) return true;
20     return balancedBlackHeight(h)!=-1;
21 }
22
23 public boolean isRedBlackBST(Node h) {
24     return is23(h) && isBalanced(h);
25 }
```

□

Exercice 5.3. Donner une méthode qui reçoit en entrée un tableau trié de n éléments et construit un arbre rouge-noir contenant les mêmes éléments en temps $O(n)$.

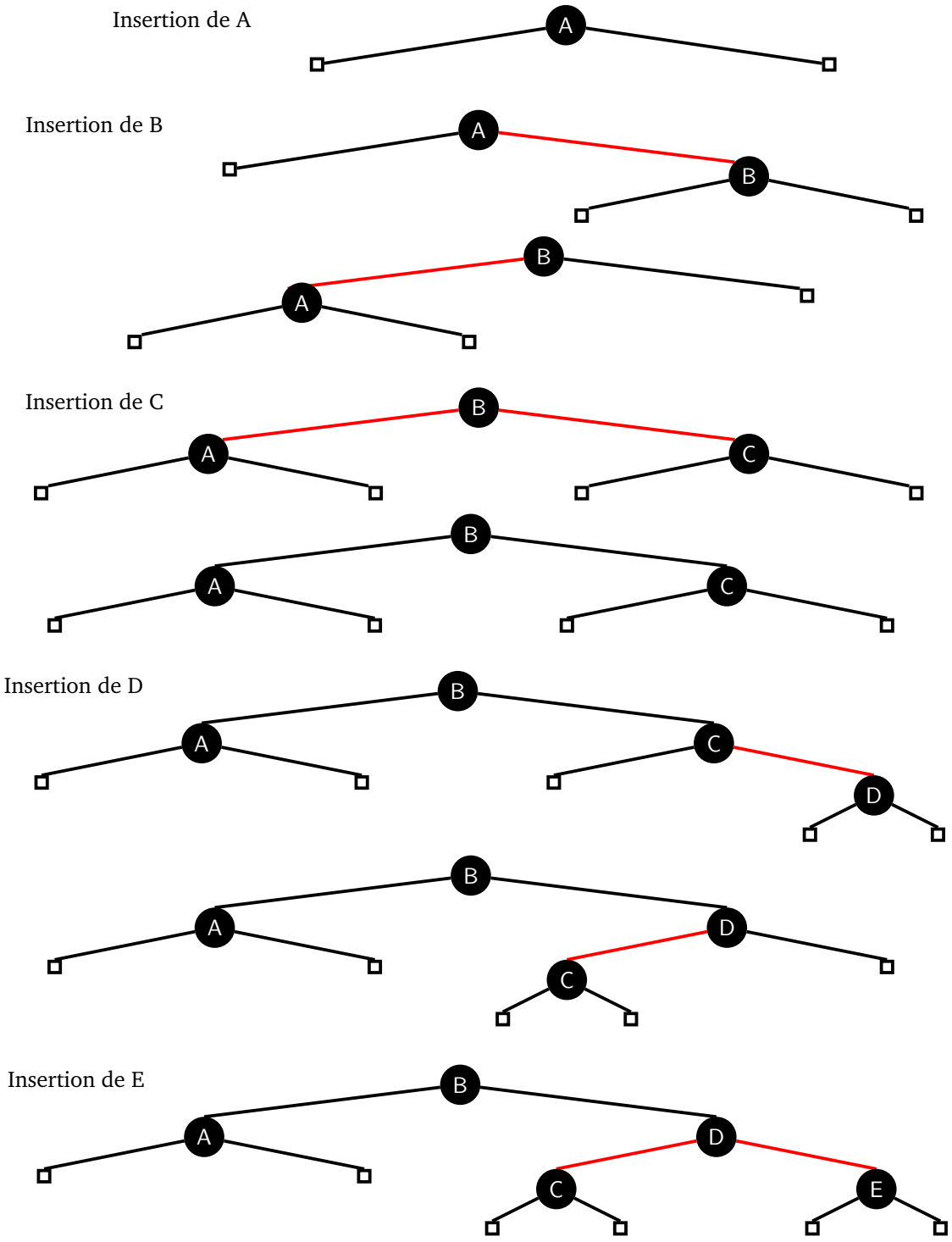


FIGURE 17 – Ajout de nœuds (1/4).

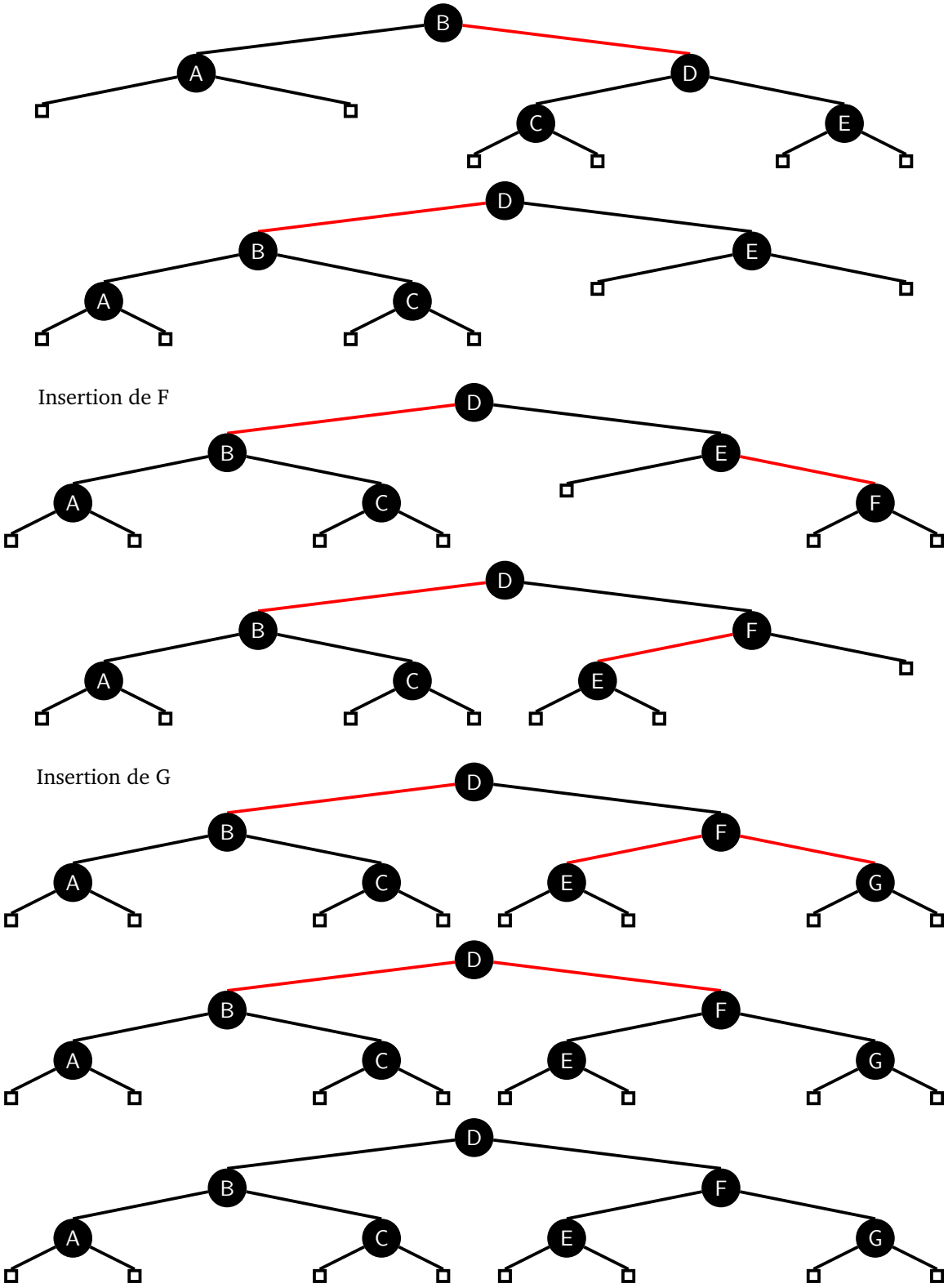


FIGURE 18 – Ajout de nœuds (2/4).

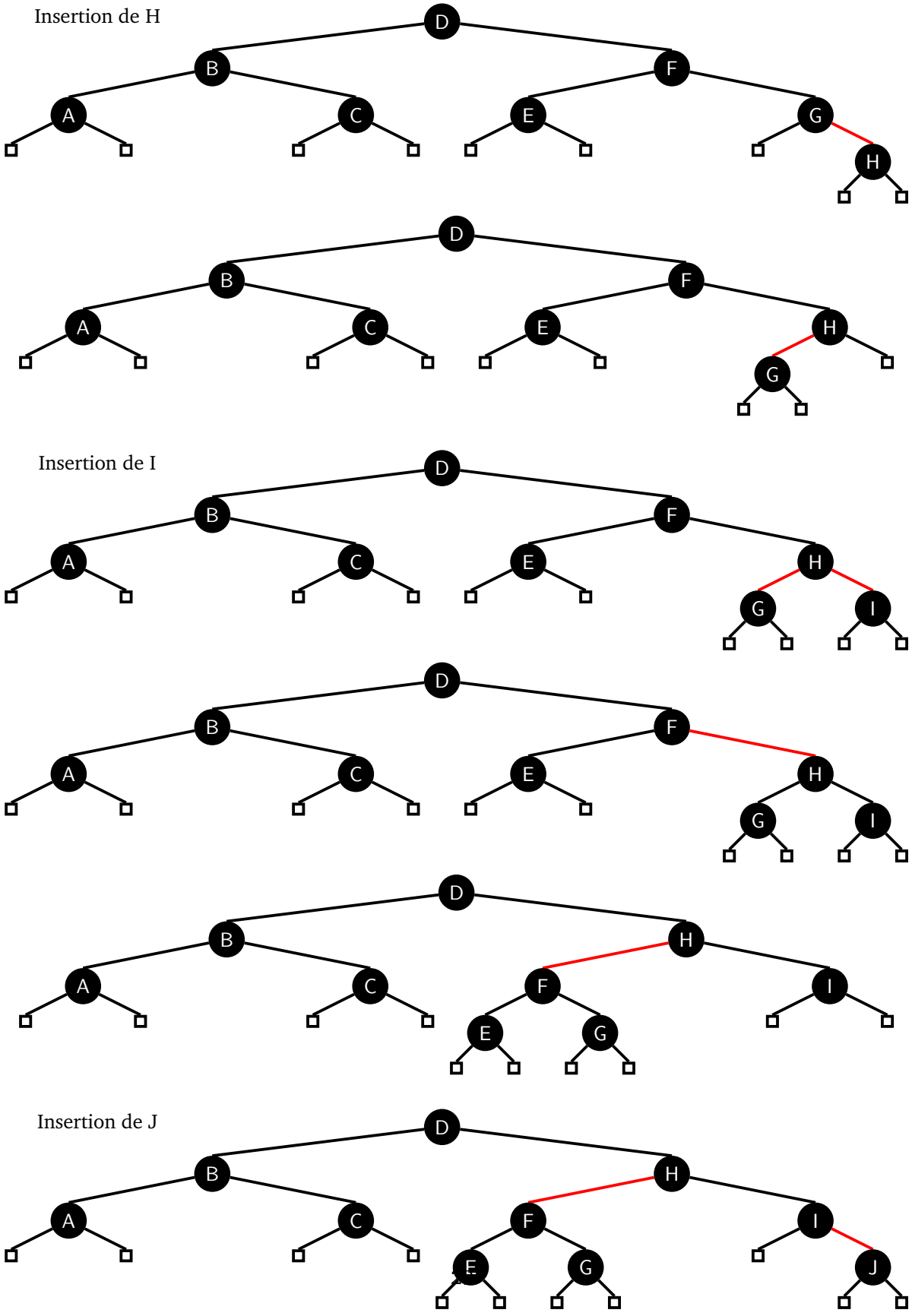


FIGURE 19 – Ajout de nœuds (3/4).

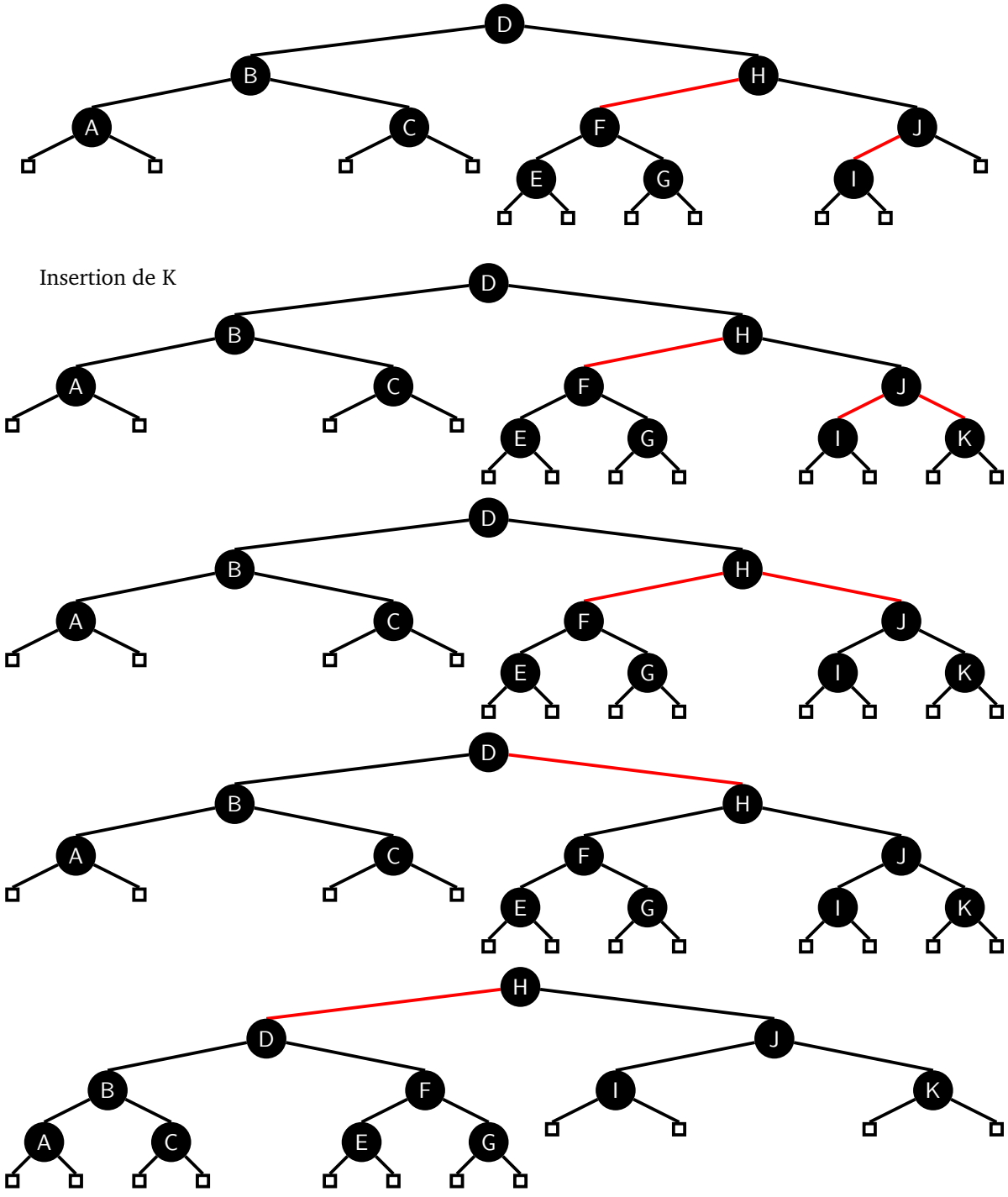


FIGURE 20 – Ajout de nœuds (4/4).

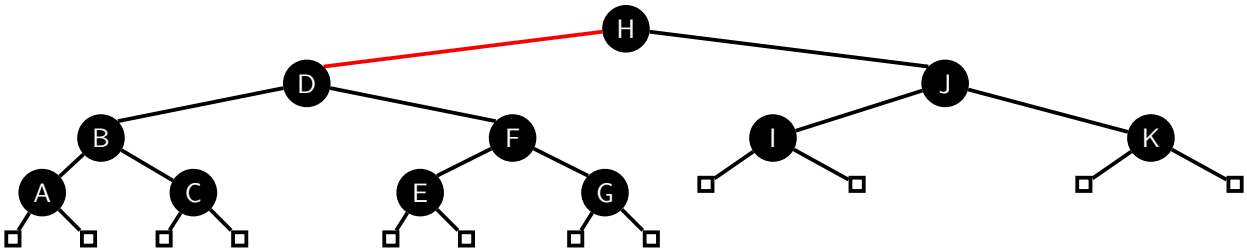


FIGURE 21 – Visualisation 2-3.

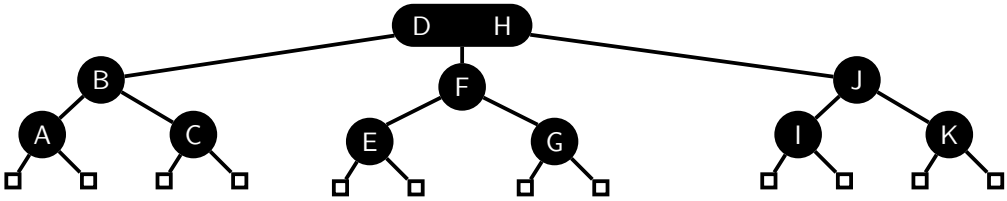


FIGURE 22 – Arbre 2-3 équivalent.

Résolution. On insère tout d'abord la médiane du tableau en tant que racine de l'arbre. Cette médiane divise les éléments restants en deux. On poursuit l'insertion en insérant tout d'abord la médiane de la moitié gauche, puis la médiane de la moitié droite. (Si la taille d'un sous-tableau est paire il existe 2 médianes, on prendra dans ce cas la plus grande des deux valeurs, ce qui assurera pour les feuilles l'aspect left-leaning sans devoir effectuer de rotations.) Les éléments restants sont à présent regroupés en 4 ensembles séparés par les 3 éléments déjà ajoutés. On peut à nouveau insérer les médianes de ces sous-ensembles. On répète ces actions jusqu'à ce que tous les éléments soient insérés. Lors de cette construction on colore les nœuds de hauteur maximale (i.e., les nœuds sans enfants) en rouge et les autres en noir. Si il y a des conflits de couleur on peut les faire remonter par niveau en au plus $O(n)$ opérations : le nombre de conflits par niveau est borné par le nombre de nœuds à chaque niveau (au plus 2^h nœuds par hauteur h), le nombre de niveaux est borné par le nombre de nœuds ($\log n$) et on peut faire remonter un conflit d'un seul niveau en temps constant (à l'aide d'un color flip).

La Figure 23 illustre l'arbre obtenu à partir d'un tableau trié contenant les lettres de A à K.

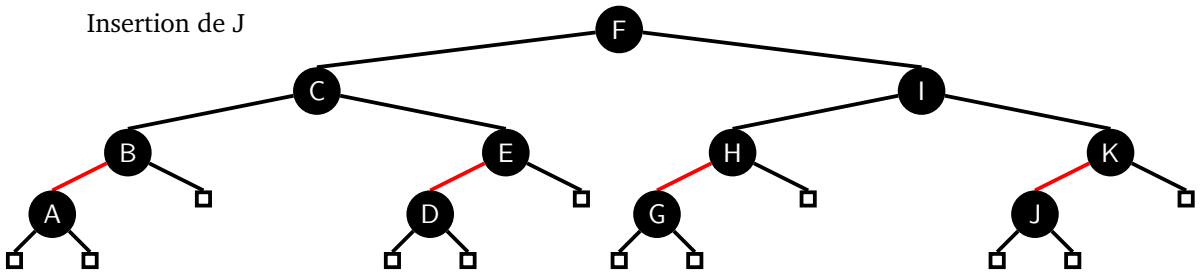


FIGURE 23 – Arbre construit à partir d'un tableau trié contenant les lettres de A à K.

```
1 class RBNode {
2     public static final int BLACK = 0;
3     public static final int RED = 1;
4     Comparable key;
5     int color;
```

```
    RBNode left, right;

    public RBNode(Comparable key, int color) {
        this.key = key;
        this.color = color;
        left = right = null;
    }

    public static RBNode createFromSortedArray(Comparable[] values, int lo, int hi,
        String offset) {
        if (lo>hi) return null;

        int mid = (int)Math.ceil((lo+hi+1)/2);

        System.out.println(offset+String.format("Adding %s (%s)",values[mid],lo==hi?
            "RED" : "BLACK"));
        RBNode mid_node = new RBNode(values[mid], lo==hi? RED : BLACK);
        mid_node.left = createFromSortedArray(values,lo,mid-1,offset+"\t");
        mid_node.right = createFromSortedArray(values,mid+1,hi,offset+"\t");

        if ((mid_node.left!=null) && (mid_node.left.color == RED) && (mid_node.right
            !=null) && (mid_node.right.color == RED)){
            System.out.println(offset+String.format("flipping %s to red and children %s
                and %s to black",mid_node.key,mid_node.left.key,mid_node.right.key));
            mid_node.left.color = BLACK;
            mid_node.right.color = BLACK;
            mid_node.color = RED;
        }
        return mid_node;
    }
    public static RBNode createFromSortedArray(Comparable[] values, int lo, int hi) {
        return createFromSortedArray(values,lo,hi,"");
    }

    public static void main(String[] argv){
        // String[] values = {"A","B","C","D","E","F"};
        String[] values = {"A","B","C","D","E","F","G"};
        createFromSortedArray(values,0,values.length-1);
    }

}
```

□

Exercice 5.4. Connexité du graphe des rotations. Démontrer que tout arbre binaire de recherche peut être transformé en n'importe quel autre arbre binaire de recherche sur les mêmes clés en n'utilisant que des rotations.

Résolution. On prend la plus petite clef du premier arbre et on la met à la racine grâce à des rotations successives vers la droite. Ensuite, on procède récursivement avec le sous-arbre de droite, jusqu'à obtenir un arbre de hauteur N (les liens vers la gauche étant tous nuls). On procède de la même manière avec le

deuxième arbre. Les figures 24 à 26 illustrent un tel processus pour une arbre contenant les lettres de A à E. □

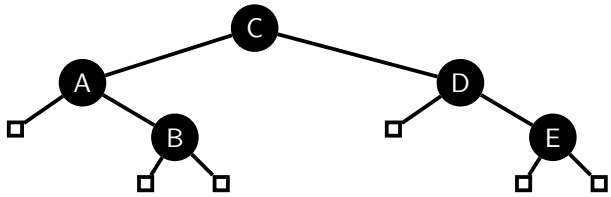


FIGURE 24 – Arbre de départ.

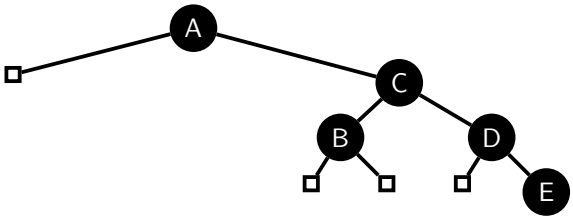


FIGURE 25 – Arbre obtenu après rotation vers la droite au nœud C.

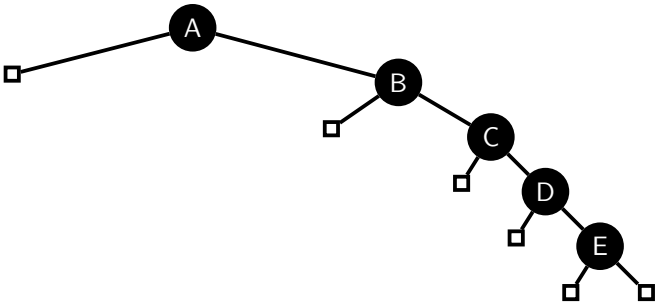


FIGURE 26 – Arbre obtenu après une nouvelle rotation vers la droite au nœud C.

Exercice 5.5 (Treaps)**.** On considère une structure d'arbre binaire dans laquelle chaque nœud contient une clé et une priorité. L'arbre est ordonné comme un arbre binaire de recherche par rapport aux clés, et comme un tas par rapport aux priorités (la racine de chaque sous-arbre a la plus grande priorité du sous-arbre).

1. Montrer qu'une telle structure existe toujours et est unique, quel que soit le choix des paires clé, priorité.
2. Supposons qu'on assigne une valeur aléatoire uniforme aux priorités. Quelle est la forme de l'arbre binaire de recherche ? Que peut-on dire de la profondeur moyenne d'un nœud ?
3. Donner un algorithme d'insertion dans un treap.

Résolution. Nous considérons ici les ensembles totalement ordonnés $\mathcal{X} = \{x_1, \ldots, x_n\}$ et $\mathcal{Y} = \{y_1, \ldots, y_n\}$ contenant respectivement les clefs et les propriétés existantes. Notons $\mathcal{V} \subset \mathcal{X} \times \mathcal{Y}$ l'ensemble des sommets où chaque clef et chaque priorité n'apparaît qu'une unique fois. Sans perte de généralisation, nous pouvons considérer que $x_1 < x_2 < \ldots < x_n$ et que $y_1 < y_2 < \ldots < y_n$.

1. À chaque clef x correspond une et une seule priorité y. On peut montrer qu'il existe un et un seul `treap` contenant $\mathcal{V}$. En effet, considérons $v^* = (x^*, y^*) \in \mathcal{V}$ tel que y^* est l'élément maximal de $\mathcal{Y}$. Celui-ci se trouve obligatoirement à la racine du `treap`. Ensuite, considérons les deux ensembles de sommets suivants :

$$\mathcal{V}_L = \{v = (x, y) \in \mathcal{V} \text{ s.t. } x < x^*\} \qquad \text{et} \qquad \mathcal{V}_R = \{v = (x, y) \in \mathcal{V} \text{ s.t. } x > x^*\}$$

   Par induction, on peut créer les *uniques* `treaps` de chacun de ces deux ensembles et les faire devenir respectivement les enfants de gauche et de droite de v^*.

2. Les `treaps` à priorité aléatoire uniforme sont en moyenne bien équilibrés. En effet, si on considère la racine de priorité maximale, les deux ensembles de clefs respectivement plus petites et plus grandes sont en moyenne de même taille. De même, on peut dire la même chose pour n'importe quel nœud de l'arbre (cfr point 1).

   On peut montrer que la profondeur moyenne d'un nœud est $O(\log n)$ où n est le nombre de nœuds dans le `treap`. En effet, considérons les clefs par ordre croissant et utilisons i et j pour référer aux positions de deux clefs dans cet ordre (cfr figure 27). Nous souhaitons analyser la profondeur du

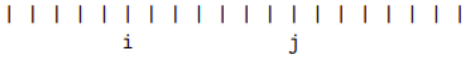


FIGURE 27 – Deux clefs dans ensemble ordonné

   nœud i dans l'arbre et nous devons donc savoir combien d'ancêtres a le nœud i. Nous utiliserons la variable aléatoire A_i^j pour indiquer si le nœud j est un ancêtre du nœud i. Notons que j peut être un ancêtre de i indépendamment de l'ordre de i et de j. La profondeur moyenne du nœud i vaut donc :

$$E[\text{profondeur de i dans T}] = E[\sum_{j=1}^{n} A_i^j] = \sum_{j=1}^{n} E[A_i^j]$$

   Pour analyser A_i^j considérons les $|j - i| + 1$ clefs se trouvant entre i et j (inclus) dans l'ensemble ordonné des clefs. Elles seules ont une influence sur le fait que j soit un ancêtre de i ou non. Considérons trois cas :
   — L'élément i a la priorité la plus grande.
   — Un des éléments k au centre a la priorité la plus grande.
   — L'élément j a la priorité la plus grande.
   Dans le premier cas, j ne peut être un ancêtre de i (qui a une priorité plus grande) et $A_i^j = 0$. Dans le deuxième cas, on a également que $A_i^j = 0$. En effet, supposons que cela ne soit pas le cas et que j soit un ancêtre de i. Alors j est également un ancêtre de k, ce qui n'est pas possible car k a une priorité plus grande. Dans le troisième cas, j est un ancêtre de i et $A_i^j = 1$. En effet, séparer i de j nécessiterait une clef intermédiaire à priorité plus haute, ce qui n'est pas possible car j a la plus grande priorité. On conclut donc que j est un ancêtre de i si et seulement si j a la plus grande priorité parmi toutes les clefs se situant entre i et j (inclusivement).
   Puisque les priorités sont aléatoires, il y a une probabilité de 1 sur $|j - i| + 1$ que $A_i^j = 1$ et $E[A_i^j] =$

$\frac{1}{|j-i|+1}$. Dès lors :

$$\begin{aligned} E[\text{profondeur de i dans T}] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\ &= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= H_i - 1 + H_{n-i+1} - 1 \\ &< 2\ln n \\ &= O(\log n) \end{aligned}$$

où on a utilisé le nombre harmonique $H_n = \sum_{i=1}^{n} \frac{1}{i}$. Ce nombre a les bornes suivantes : $\ln\ n < H_n < \ln\ n + 1$ [5].

Nous pouvons également montrer cela comme suit : la profondeur moyenne des sommets est donnée par $\frac{1}{n}S(n)$ où $S(n)$ est la somme de la profondeur des sommets. Si la racine est le $k$ème sommet, alors il y a $k-1$ sommets dans le sous-arbre gauche et $n-k$ sommets dans le sous-arbre droit, et $S(n) = k-1+S(k-1)+S(n-k)+n-k$. En effet la somme des profondeurs du sous-arbre gauche vaut $S(k-1)+k-1$ et la somme des profondeurs du sous-arbre droit vaut $S(n-k)+n-k$. Dès lors :

$$S(n) = n-1+S(k-1)+S(n-k).$$

En moyenne, puisque la racine a une probabilité de $\frac{1}{n}$ d'être chacun des sommets (entre 1 et n), nous avons :

$$S(n) = \frac{1}{n}\sum_{k=1}^{n}(n-1+S(k-1)+S(n-k)) = n-1+\frac{2}{n}\sum_{k=0}^{n-1}S(k).$$

Nous avons déjà vu cette relation de récurrence lors de l'analyse de complexité moyenne de `quicksort`, et nous savons donc que $S(n) \sim 2n\ln n$. Dès lors nous pouvons déduire que la profondeur moyenne des sommets est $\frac{1}{n}S(n) \sim 2\ln n$.

3. L'algorithme d'insertion dans un `treap` consiste à insérer le nouveau nœud en fonction sa clef, de la même manière que dans un arbre binaire de recherche. Ensuite, on vérifie que la priorité du nouveau nœud est bien inférieure à celle de son parent. Si ce n'est pas le cas, on procède à une rotation vers la gauche ou vers la droite au niveau du parent, selon que le nouveau nœud se trouve respectivement à droite ou à gauche de son parent. Après cette rotation, on compare à nouveau la priorité du nouveau nœud avec celle de son nouveau parent et on procède à une rotation si nécessaire. On procède ainsi itérativement jusqu'à ce que la priorité du nouveau nœud soit inférieure à celle de son parent ou que le nœud se trouve à la racine de l'arbre.

```
1  class Treap {
2
3      /* Classe contenant l'enfant de gauche et de droite du noeud actuel ainsi que
           la clef et la priorité */
4      class Node {
5          int key;
6          int priority;
7          Node left, right;
8
9          public Node(int item, int p) {
10             key = item;
```

5. On peut voir que $H_n \approx \ln\ n$ en considérant l'intégrale $\int_0^n \frac{1}{x}\mathrm{d}x$

```
11              priority = p;
12              left = right = null;
13          }
14      }
15
16      // Racine du Treap
17      Node root;
18
19      // Constructeur
20      Treap() {
21          root = null;
22      }
23
24      // Cette méthode utilise insertRec()
25      void insert(int key, int priority) {
26          root = insertRec(root, key, priority);
27      }
28
29      /* Une fonction récursive qui insère un noeud dans un Treap */
30      Node insertRec(Node root, int key, int priority) {
31
32          /* si le sous-treap est nul, insérer un nouveau noeud */
33          if (root == null) {
34              root = new Node(key, priority);
35              return root;
36          }
37
38          /* Sinon, procéder par récurrence */
39          if (key < root.key) {
40              root.left = insertRec(root.left, key, priority);
41              if (root.priority < root.left.priority) {
42                  root = rotateRight(root);
43              }
44          }
45          else if (key > root.key) {
46              root.right = insertRec(root.right, key, priority);
47              if (root.priority < root.right.priority) {
48                  root = rotateLeft(root);
49              }
50          }
51
52          /* retourne le pointeur du noeud (qui peut avoir été modifié) */
53          return root;
54      }
55
56      /* Faire une rotation gauche */
57      Node rotateLeft(Node h)
58      {
59          Node x = h.right;
60          h.right = x.left;
61          x.left = h;
```

```
62          return x;
63      }
64
65      /* Faire une rotation droite */
66      Node rotateRight(Node h)
67      {
68          Node x = h.left;
69          h.left = x.right;
70          x.right = h;
71          return x;
72      }
73  }
```

□

Séance 6 — Parcours de graphes

Rappel (Graphes et chemins). *Un* graphe *est une paire $G = (\mathcal{V}, \mathcal{E})$ de sommets et d'arêtes (i.e. $\mathcal{E} \subseteq \binom{\mathcal{V}}{2}$ si G est non dirigé, et $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ si G est dirigé). On note V la cardinalité de $\mathcal{V}$ et E la cardinalité de $\mathcal{E}$.*

Un chemin *P dans un graphe $G = (\mathcal{V}, \mathcal{E})$ est une suite ordonnée de sommets $P = (v_1, \ldots, v_n)$ tels que v_i et v_{i+1} sont adjacents pour tout $0 < i < n$. Si v_1 et v_n sont adjacents, alors P est un* cycle. *Si G n'admet pas de cycle, alors il est dit* acyclique.

Un chemin (ou cycle) est dit eulérien *s'il passe une unique fois par chaque arête de G, et est dit* hamiltonien *s'il passe une unique fois par chaque sommet.*

Un graphe $G = (\mathcal{V}, \mathcal{E})$ est biparti *si on peut partitionner son ensemble de sommets $\mathcal{V}$ en deux sous-ensembles disjoints $\mathcal{U}$ et $\mathcal{W}$ tels que pour toute paire de sommets adjacents (v_i, v_j), soit $v_i \in \mathcal{U}$ et $v_j \in \mathcal{W}$; soit $v_i \in \mathcal{W}$ et $v_j \in \mathcal{U}$.*

Exercice 6.1. Donner un algorithme de complexité linéaire pour les problèmes suivants, sur un graphe non-dirigé G donné en entrée :

1. Le graphe G est-il acyclique ?
2. Le graphe G est-il biparti ?
3. Le graphe G possède-t-il un cycle eulérien ?

Résolution.

1. Trouver un cycle dans un graphe *non-dirigé* peut être réalisé par un parcours en profondeur (DFS). En effet en partant d'un nœud quelconque $v \in \mathcal{V}$, un DFS marquera tous les sommets accessibles depuis v. De plus, si un sommet $w \in \mathcal{V}$ est marqué deux fois, c'est qu'il existe deux chemins distincts liant v et w. Dès lors, si u est le dernier ancêtre de w commun dans ces deux chemins, alors il existe un cycle dirigé contenant u et w. En appliquant un DFS sur chaque nœud n'ayant pas encore été marqué, toutes les possibilités de cycle sont explorées.
   En procédant de la sorte, chaque arête est visitée au plus une fois, et il en est de même pour les sommets puisque la recherche d'un cycle s'arrête dès qu'un sommets marqué deux fois est rencontré.

```
1  private boolean dfs(Graph G, int v, boolean marked[], int parent) {
2      if(marked[v])
3          return true;
4      marked[v] = true;
5      for(int w : G.adj(v))
6          if(w != parent && dfs(G, w, marked, v))
7              return true;
8      return false;
9  }
10
11 public boolean detectCycle(Graph G) {
12     boolean marked[] = new boolean[G.V()];
13     for(int v = 0; v < G.V(); ++v)
14         if(!marked[v] && dfs(G, v, marked, -1))
15             return true;
16     return false;
17 }
```

2. Déterminer si un graphe est biparti peut également être déterminé par un DFS : on détermine deux labels (e.g. deux couleurs) qui serviront à marquer chaque sommet visité selon son appartenance à $\mathcal{U}$

ou $\mathcal{W}$.

Pour chaque nœud n'ayant pas encore été marqué, on lui assigne arbitrairement un label (soit celui correspondant à $\mathcal{U}$, soit celui correspondant à $\mathcal{W}$), et on assigne à chacun de ses enfants le label correspondant à l'autre sous-ensemble. Si à un moment on tente d'assigner le label de $\mathcal{U}$ à un sommet déjà marqué par le label de $\mathcal{W}$ (ou inversement), alors le graphe G n'est pas biparti. Si tous les sommets ont été visités (et ont donc un label assigné) sans qu'il n'y ait eu de sommet à deux labels, alors le graphe G est biparti, et les labels donnent la partition en $\mathcal{U}$ et $\mathcal{W}$.

Par le même raisonnement que dans la sous-question précédente, on se convainc aisément que le nombre d'opérations est linéaire en V et E.

```
private boolean labelNodes(Graph G, int v, int labels[], int parent, int label) {
    int otherLabel = (label == 1) ? 2 : 1;
    if(labels[v] == otherLabel)
        return false;
    else if(labels[v] == label)
        return true;
    labels[v] = label;
    for(int w : G.adj(v))
        if(w != parent && !labelNodes(G, w, labels, v, otherLabel))
            return false;
    return true;
}

public boolean isBipartite(Graph G) {
    int labels[] = new int[G.V()];
    for (int v = 0; v < G.V(); ++v)
        if (labels[v] == 0 && !labelNodes(G, v, labels, -1, 1))
            return false;
    return true;
}
```

3. Un graphe non-dirigé est eulérien si et seulement si tous ses nœuds sont de degré pair, et tous les sommets de degré strictement positif appartiennent à la même composante connexe, il est donc suffisant de tester ces deux conditions.

   Déterminer si les degrés sont pairs est trivial : il suffit d'itérer sur chaque sommet et de compter le nombre de sommets qui y sont adjacents. Cela requiert un nombre d'opérations en $O(V)$ (en supposant que le degré d'un sommet se récupère en temps constant).

   Afin de déterminer si tous les sommets de degré strictement positif appartiennent à la même composante connexe, il suffit de choisir arbitrairement un sommet v tel que $\deg(v) > 0$, et de marquer tous les sommets accessibles depuis $v \in \mathcal{V}$ à l'aide d'un DFS. Si tous les sommets non-marqués sont de degré nul, alors la condition est satisfaite. À l'inverse, s'il existe un sommet $w \in \mathcal{V}$ non marqué et tel que $\deg(w) \gneqq 0$, alors la condition n'est pas satisfaite.

   Dans le pire des cas, chaque arête est visitée exactement une fois, et chaque sommet également. Dès lors déterminer si G est eulérien prend un temps linéaire en V et E.

```
private void mark(Graph G, int v, boolean marked[]) {
    marked[v] = true;
    for(int w : G.adj(v))
        if(!marked[w])
            mark(G, w, marked);
}
```

```
7
8  private boolean isConnected(Graph G) {
9      boolean marked[] = new boolean[G.V()];
10     for(int v = 0; v < G.V(); ++v)
11         if(G.adj(v).size > 0)
12             break;
13     if(v == G.V())
14         return true;
15     mark(G, v, marked);
16     for(int v = 0; v < G.V(); ++v)
17         if(!marked[v] && G.adj(v).size > 0)
18             return false;
19     return true;
20 }
21
22 public boolean isEulerian(Graph G) {
23     if(!isConnected(G))
24         return false;
25     for(int v = 0; v < G.V(); ++v)
26         if(G.adj(v).size & 1)
27             return false;
28     return true;
29 }
```

□

Exercice 6.2. Que peut-on dire de la complexité des problèmes suivants ?

1. Étant donnés deux graphes G et H comportant le même nombre de sommets, est-il vrai que G et H sont identiques, à une permutation des noms des sommets près ?
2. Étant donné un graphe G, possède-t-il un cycle hamiltonien ?

Résolution. Afin de répondre à cette question, il nous faut définir (au moins approximativement) quelques notions algorithmiques qui seront vues de manière plus rigoureuse et plus complète dans le cours de *calculabilité et complexité* (donné en MA1).[6]

Définition 6.1. On note P l'ensemble des problèmes qui peuvent être résolus en temps polynomial (i.e. un problème Q est dans P lorsque $\exists k \in \mathbb{N}$ tel qu'il existe un algorithme pour résoudre Q en temps $O(n^k)$).

Notons bien que l'on ne demande pas que tous les algorithmes pour résoudre un tel problème soient polynomiaux en temps, mais que l'on demande bien qu'il existe un algorithme de temps polynomial dans le pire des cas.

Les trois problèmes posés à la question précédente (déterminer si un graphe est acyclique, biparti, ou eulérien) appartiennent tous à P puisque des algorithmes linéaires (donc polynomiaux) existent pour ces trois problèmes.

Définition 6.2. On note NP l'ensemble des problèmes dont la solution est *vérifiable* en temps polynomial.

6. Afin d'arriver aux définitions classiques de ces notions, il faudrait parler d'automates et de machines de Turing, ce qui dépasse très largement les objectifs de ce cours d'*algorithmique*.

Décider si un graphe possède un cycle est un problème dans NP : il est facile de vérifier que la réponse est OUI si on se donne une suite de sommet formant un cycle. Mais le problème du cycle hamiltonien est aussi dans NP : il est facile de vérifier qu'une permutation des sommets forme un cycle hamiltonien. Tous les problèmes auxquels vous avez été confrontés jusqu'ici sont probablement dans NP, mais cette classe de problème ne contient pas tous les problèmes étudiés en théorie de la complexité.

NOTE. Il est important de noter que NP **ne veut pas dire** *impossibe à résoudre en temps polynomial*, mais bien *possible à résoudre en temps polynomial non-déterministe* (peu importe ce que cela veut dire précisément).

On peut montrer que $\mathsf{P} \subseteq \mathsf{NP}$, mais une grande (très grande) question en informatique théorique est la suivante : *est-ce que* $\mathsf{P} = \mathsf{NP}$*?* Cette question est tellement importante qu'elle fait partie des 7 *Millenium Problems* de l'institut Clay, i.e. quiconque arriverait à prouver **rigoureusement** que $\mathsf{P} = \mathsf{NP}$ ou que $\mathsf{P} \neq \mathsf{NP}$ serait récompensé d'un **million** d'USD. [7]

Définition 6.3. Un problème Q est dit NP-complet s'il appartient à NP et si tout problème R de NP peut être *réduit* à Q en temps polynomial, i.e. si toute instance du problème R peut être *transformée* en une instance équivalente de Q en temps polynomial.

Notons que si il existe un problème $P \in \mathsf{P}$ qui est NP-complet, alors $\mathsf{P} = \mathsf{NP}$.

Définition 6.4. un problème Q dans NP est dit NP-intermédiaire s'il n'est pas NP-complet, et s'il n'est pas dans P.

Notons maintenant que si un problème NP-intermédiaire existe, alors $\mathsf{P} \neq \mathsf{NP}$. En 1975, Richard Ladner a démontré que si $\mathsf{P} \neq \mathsf{NP}$, alors il doit exister des problèmes NP-intermédiaires.

Cycle hamiltonien Le problème du cycle hamiltonien est un problème NP-complet. Afin de démontrer qu'un problème P est NP-complet, deux méthodes sont possibles : soit par la définition, on donne une réduction en temps polynomial pour chaque problème Q de NP vers P ; soit on prend un autre problème NP-complet Q, et on montre que Q peut être réduit vers P en temps polynomial. C'est cette seconde approche qui est utilisée pour montrer que le problème de cycle hamiltonien est NP-complet. Notons que pour pouvoir utiliser cette seconde approche, il est nécessaire d'avoir au préalable un problème montré NP-complet.

Cela implique que si $\mathsf{P} \neq \mathsf{NP}$, on sait que déterminer si un graphe contient un cycle hamiltonien est un problème *difficile*, et même un des problèmes *les plus difficiles* en informatique. Mais ce n'est pas pour autant qu'il est impossible de le résoudre. Par exemple, un simple backtracking permet de trouver un cycle hamiltonien dans un graphe G s'il en existe un.

```
1  private boolean _hamiltonian(Graph G, int v, boolean marked[], int nbMarked) {
2      if(v == 0 && nbMarked == G.V())
3          return true;
4      for(int w : G.adj(v)) {
5          if(marked[w])
6              continue;
7          marked[w] = true;
8          if(_hamiltonian(G, w, marked, nbMarked+1))
9              return true;
10         marked[w] = false;
11     }
12     return false;
```

7. Notons que toute personne désirant chercher la solution devrait regarder les avancées sur le sujet : https://www.win.tue.nl/~gwoegi/P-versus-NP.htm.

```
}

public boolean hamiltonian(Graph G) {
    boolean marked[] = new boolean[G.V()];
    return _hamiltonian(G, 0, marked, 0);
}
```

Dans le pire des cas, ce backtracking parcourt tous les cycles du graphe G, dont le nombre peut être proche de $n!$.

Isomorphisme de graphes Le cas de l'isomorphisme de deux graphes est plus compliqué : en effet, à défaut de trouver un algorithme polynomial pour le résoudre, il a pendant un temps été pensé que ce problème était NP-complet également. Cependant jusqu'à présent, aucune preuve n'a été trouvée, malgré le nombre de chercheurs ayant essayé. De plus, en 1987, Uwe Schöning a donné un argument qui permet de penser que le problème d'isomorphisme de graphes n'est en effet pas NP-complet.

Aujourd'hui, certains pensent que ce problème est NP-intermédiaire, mais cela n'a toujours pas été démontré non plus. Actuellement, seuls des algorithmes exponentiels dans le pire des cas sont connus pour déterminer si deux graphes sont isomorphes.

Notons que ce problème est toujours un problème actif en théorie des graphes et en théorie de la complexité. En effet, contrairement au problème précédent, montrer que ce problème peut être résolu en temps polynomial ne permettrait pas de résoudre $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, mais s'il est possible de montrer qu'il est en effet NP-intermédiaire, alors on saura que $\mathsf{P} \neq \mathsf{NP}$.

En 1983, László Babai et Eugene Luks ont proposé un algorithme en temps $2^{O(\sqrt{n \log n})}$ pour résoudre le problème d'isomorphisme de graphes. Cet algorithme est le meilleur connu à ce jour, et est donc encore utilisé.

Plusieurs cas particuliers de ce problème peuvent être résolus en temps polynomial : e.g. si G et H sont deux graphes planaires, alors on peut tester s'ils sont isomorphes en temps linéaire ; si G et H sont des arbres, alors en particulier ils sont planaires donc à nouveau on peut résoudre le problème en temps linéaire.

Plus récemment – en 2016 – László Babai a proposé une preuve montrant que ce problème peut être résolu en temps quasipolynomial (i.e. en temps $2^{O((\log n)^c)}$). Bien que ce papier n'ait pas encore été accepté, **si la preuve est correcte**, alors le problème ne peut pas être NP-complet, sauf si tous les problèmes NP-complets peuvent être résolus en temps quasipolynomial également (notons que cela n'impliquerait pas nécessairement que le problème est dans P).

La conclusion à tirer de cet exercice est la suivante : en cours d'algorithmique, vous découvrez des algorithmes et des structures de données qui permettent de résoudre plusieurs problèmes de manière très intéressante (i.e. des algorithmes polynomiaux). Mais certains problèmes sont en réalité difficiles à résoudre, voire très difficile à résoudre (sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$). De plus, contrairement à une impression que vous pourriez avoir suite à vos cours d'algorithmique, tous les problèmes classiques ne sont pas encore résolus. Certains problèmes font encore l'objet de recherches approfondies pour tenter de déterminer leur complexité. □

*** Exercice 6.3** (Graphes 2-arête-connexes)**.** Un pont dans un graphe connexe est une arête telle que si on l'enlève, le graphe est divisé en deux composantes connexes. Un graphe sans pont est dit *2-arête-connexe*. Donner un algorithme qui détermine si un graphe donné est 2-arête-connexe.

Résolution. L'algorithme ci-dessous permet de trouver les ponts dans un graphe. Il utilise la quantité clef `low[v]` pour un sommet `v`, qui est le préordre dfs minimum de `v` et de l'ensemble des sommets `w` tels qu'il

existe une arête w-x, où w est un ancêtre de v et x est un descendant de v (voir figure 28). Notez que cet algorithme suppose que le graphe ne contient pas d'arêtes parallèles (par exemple une double arête entre deux sommets). Comme indiqué sur la figure 29, l'algorithme repère les sommets dont la valeur low correspond à leur préordre dfs, ce qui indique que le sommet est l'extrémité d'un pont.

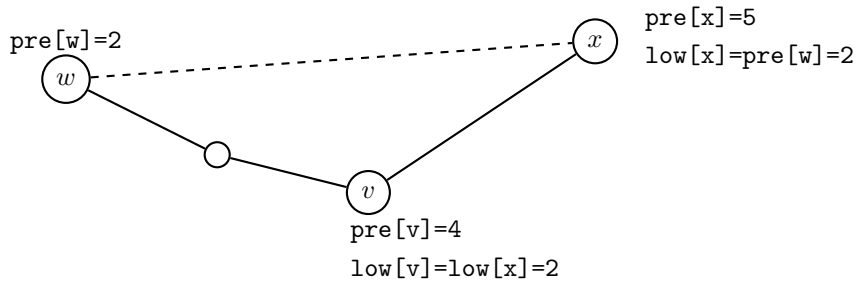


FIGURE 28 – Partie d'un graphe parcourue par dfs dans l'ordre w, v, x avec leurs valeurs de pre et low respectives.

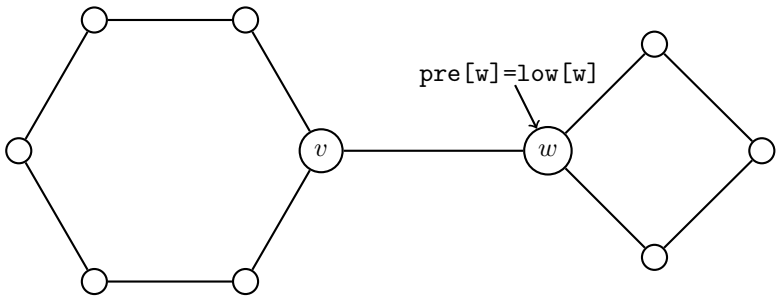


FIGURE 29 – Exemple de pont (arête v-w). La dfs se fait dans l'ordre v, w.

```java
public class Pont {
    private int ponts;    // nombre de ponts
    private int cnt;        // compteur
    private int[] pre;      // pre[v] = ordre dans lequel dfs examine v
    private int[] low;      // low[v] = préordre minimum de v et de l'ensemble des
                            // sommets w tels qu'il existe une arête w-x, ou w est un
                            // ancêtre et x un descendant de v

    public Pont(Graph G) {
        low = new int[G.V()];
        pre = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            low[v] = -1;
        for (int v = 0; v < G.V(); v++)
            pre[v] = -1;

        for (int v = 0; v < G.V(); v++)
            if (pre[v] == -1)
                dfs(G, v, v);
    }
```

```
21
22      public int composantes() { return ponts + 1; }
23
24      private void dfs(Graph G, int u, int v) {
25          pre[v] = cnt++;
26          low[v] = pre[v];
27          for (int w : G.adj(v)) {
28              if (pre[w] == -1) {
29                  dfs(G, v, w);
30                  low[v] = Math.min(low[v], low[w]);
31                  if (low[w] == pre[w]) {
32                      StdOut.println(v + "-" + w + " est un pont");
33                      ponts++;
34                  }
35              }
36
37              // mise à jour de low - ignore pour le reste cette arête
38              else if (w != u)
39                  low[v] = Math.min(low[v], pre[w]);
40          }
41      }
42
43      // test
44      public static void main(String[] args) {
45          int V = Integer.parseInt(args[0]);
46          int E = Integer.parseInt(args[1]);
47          Graph G = GraphGenerator.simple(V, E);
48          StdOut.println(G);
49
50          Pont pont = new Pont(G);
51          StdOut.println("Composantes 2-arêtes connexes = " + pont.composantes());
52      }
53
54
55  }
```

□

Exercice 6.4 (Un jeu de rôle)**.** Un monstre et un joueur se trouvent chacun sur un sommet distinct d'un graphe (simple, non dirigé). Le joueur et le monstre jouent à tour de rôle. A chaque tour, ils peuvent se déplacer sur un sommet adjacent, ou rester sur le même sommet. Donner un algorithme pour déterminer tous les sommets que le joueur peut atteindre avant le monstre. On suppose que le joueur joue en premier.

Résolution. En effectuant un parcours en largeur (breadth-first search, bfs) à partir d'un sommet s on peut en passant établir la distance de chaque autre sommet s' au sommet de départ s, qu'on dénote ici comme $d(s, s')$.

Etant donné le sommet de départ du joueur j et le sommet de départ du monstre m, on effectue un parcours en largeur à partir de j et ensuite un parcours en largeur à partir de m. On établit ainsi la distance entre ces deux sommets et n'importe quel autre sommet du graphe.

Pour déterminer quels sommets le joueur peut atteindre avant le monstre il suffit de trouver chaque sommet s' pour lequel $d(j, s') \leq d(m, s')$. Notons qu'en cas de longueur égale c'est le joueur qui l'emporte, puisqu'il joue en premier.

Le code java ci-dessous modifie le parcours en largeur afin de retourner un tableau de distances. On appelle cette fonction pour déterminer la distance entre chaque case et le joueur/monstre. On parcourt ensuite ces deux tableaux afin d'établir le joueur le plus proche de chaque sommet.

```
1  private int[] computeDistances(Graph G, int s) {
2      // parcours en largeur avec retour des distances
3      Queue<Integer> q = new Queue<Integer>();
4      q.enqueue(s);
5      boolean[] marked = new boolean[G.V()]; // valeurs par défaut = false
6      marked[s] = true;
7      int[] distTo = new int[G.V()]; // valeurs par défaut = 0
8      while (!q.isEmpty()) {
9          int v = q.dequeue();
10         for (int w : G.adj(v)) {
11             if (!marked[w]) {
12                 q.enqueue(w);
13                 marked[w] = true;
14                 distTo[w] = distTo[v] + 1;
15             }
16         }
17     }
18     return distTo;
19 }
20
21 private void printFastestPlayer(Graph G, int j, int m) {
22     int[] playerDistances = computeDistances(G,j);
23     int[] monsterDistances = computeDistances(G,m);
24
25     for(int s = 0; s<G.V(); s++) {
26         // On établi le plus proche des joueurs pour chaque sommet
27         String closest = playerDistances[s] <= monsterDistances[s] ? "joueur" : "monstre";
28         StdOut.println(String.format("Le %s peut atteindre le sommet %d en
               premier.",closest,s);
29     }
30
31 }
```

□

Séance 7 — Connexité forte

Exercice 7.1. Quelles sont les composantes fortement connexes d'un graphe dirigé acyclique ? Expliquer comment se comporte l'algorithme de Kosaraju-Sharir dans ce cas particulier.

Résolution. Les composantes fortement connexes d'un graphe dirigé acyclique correspondent aux différents sommets du graphe. En effet, s'il existait une composante fortement connexe à plusieurs sommets, il y aurait un cycle dans le graphe, ce qui contredirait l'hypothèse de départ.

Lors du deuxième parcours de l'algorithme de Kosaraju-Sharir et lors de l'appel de `dfs(G,s)`, il n'y aura pas de sommet à la fois atteignable à partir du sommet s et non marqué, autre que le sommet s lui-même. En effet, si dans le graphe G, il existe une arête dirigée du sommet a vers le sommet b, ce dernier se trouvera avant le sommet a dans le post-ordre de G, car il n'existe pas de chemin du sommet b vers le sommet a. Dès lors, b se trouvera avant a également dans le post-ordre inverse de $G^\top$. Par conséquent, lors du deuxième parcours, le sommet b sera déjà marqué lors de l'appel de `dfs(G,a)`. Cela vaut pour tous les sommets directement atteignables via le sommet a. □

Exercice 7.2. Vrai ou faux ? Justifier votre réponse par une démonstration (si vrai) ou un contre-exemple (si faux) :

1. Un postordre inverse du tranposé d'un graphe dirigé G est un postordre de G.
2. Si on inverse le rôle de G et $G^\top$ dans l'algorithme de Kosaraju-Sharir, le résultat reste correct.
3. Si l'on remplace le parcours en profondeur de la seconde phase de l'algorithme de Kosaraju-Sharir par un parcours en largeur, le résultat reste correct.

Résolution.

1. **Faux**. La Figure 30 montre un contre-exemple.

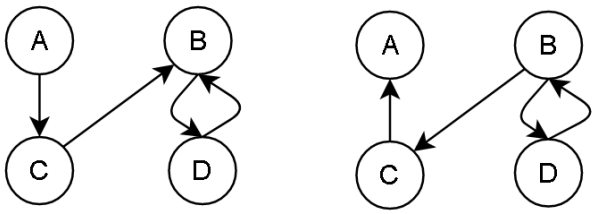


FIGURE 30 – Un des post-ordres possibles du graphe transposé (à droite) est $ADCB$ (on commence sur A, puis sur B en choisissant en premier D comme adjacent). Cependant, son inverse, $BCDA$, n'est pas un des post-ordres possibles du graphe de départ (à gauche).

2. **Vrai**. En effet les composantes fortement connexes de G et $G^\top$ sont les mêmes. Pour s'en convaincre, deux sommets $u, v \in V$ appartiennent à la même composante fortement connexe de G si et seulement si il existe un chemin $P_{u\to v} = (u^0, u^1, \dots, u^k)$ tel que $u^0 = u$ et $u^k = v$ et un chemin $P_{v\to u} = (v^0, v^1, \dots, v^\ell)$ tel que $v^0 = v$ et $v^\ell = u$. Puisque $(u^i, u^{i+1}) \in E(G)$ pour $i < k$ et $(v^j, v^{j+1}) \in E(G)$ pour $j < \ell$, on sait que $(u^{i+1}, u^i) \in E(G^\top)$ et $(v^{j+1}, v^j) \in E(G^\top)$ pour $i < k$ et $j < \ell$.
   Dès lors $P^\top_{u\to v} = (v^\ell, v^{\ell-1}, \dots, v^0)$ est un chemin de u vers v dans $G^\top$ et $P^\top_{v\to u} = (u^k, u^{k-1}, \dots, u^0)$ est un chemin de v vers u dans $G^\top$.
3. **Vrai**. En effet, l'objectif du deuxième parcours est de marquer tous les sommets `u` tels qu'il existe un chemin de `v` vers `u`. Dès lors un parcours (qu'il soit en profondeur ou en largeur) marquera tous ces sommets.

□

Exercice 7.3. Donner un algorithme de complexité linéaire pour calculer la composante fortement connexe d'un graphe G contenant un sommet donné v.

Résolution. La procédure consiste en trois étapes :
- — trouver les sommets joignables à partir du sommet v,
- — trouver les sommets qui peuvent atteindre le sommet v,
- — prendre l'intersection des deux ensembles.

La première étape peut être réalisée simplement à l'aide d'un parcours en profondeur sur G. La deuxième également, mais cette fois sur G^T. Les trois étapes étant de complexité linéaire en le nombre de sommets et d'arêtes, l'algorithme complet est également de complexité linéaire.

```java
public class StronglyConnectedComponent {

    // parcours en profondeur
    private void dfs(Digraph G, int v, boolean[] marked) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) {
                dfs(G, w, marked);
            }
    }

    // trouve sommets fortement connectés au sommet v
    public boolean[] findStronglyConnected(Digraph G, int v) {
        boolean[] stronglyConnected = new boolean[G.V()];
        boolean[] marked1 = new boolean[G.V()]; // sommets joignables via sommet v
        boolean[] marked2 = new boolean[G.V()]; // sommets qui peuvent atteindre sommet v
        dfs(G, v, marked1); // remplit vecteur marked1
        dfs(G.reverse(), v, marked2); // remplit vecteur marked2
        for (int i = 0; i < G.V(); i++)
            stronglyConnected[i] = marked1[i] && marked2[i];
        return stronglyConnected;
    }

    //test
    public static void main(String[] args) {
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);
        int v = Integer.parseInt(args[2]);
        Digraph G = DigraphGenerator.simple(V, E);
        StdOut.println(G);

        StronglyConnectedComponent SCC = new StronglyConnectedComponent();
        boolean[] stronglyConnected = SCC.findStronglyConnected(G, v);
        for (int i = 0; i < G.V(); i++) if (stronglyConnected[i]) StdOut.print(i + ",");
    }
}
```

□

Exercice 7.4. Un *cycle impair* est un cycle de longueur impaire. Donner un algorithme de complexité linéaire pour décider si un graphe dirigé possède un cycle (dirigé) impair. *Indice : un graphe (non-dirigé) est biparti si et seulement s'il ne possède aucun cycle impair.*

Résolution. Si un cycle dirigé impair existe, il sera contenu dans une des composantes fortement connexes de G.

Prenons G' une composante fortement connexe du digraphe G. On peut vérifier en temps linéaire en la taille de G' si celui-ci ne contient pas de cycle **non-dirigé** impair (en vérifiant s'il est biparti, cfr. séance 6). Notons chaque cycle dirigé est également un cycle non-dirigé, par conséquent, s'il n'existe pas de cycle non-dirigé impair, il n'existe pas de cycle dirigé impair. Considérons alors le cas où il existe un cycle non-dirigé impair C et montrons comment un cycle dirigé peut être construit à partir de celui-ci. Prenons la séquence de sommets couverts par le cycle non-dirigé : $a - .... - u - v - ... - z$. Dans un premier temps, remplaçons chaque arête non-dirigée par l'arc équivalent dirigé : $a \to .... \to u \leftarrow v \to ... \to z$. Puisque la composante G' est fortement connexe, pour chaque pair de sommets successifs u, v dont l'arc va dans le mauvais sens on pourrait trouver un chemin P allant de u à v par une recherche en profondeur ou en largeur. Il y a alors deux possibilités :

— Soit P est de longueur paire, on peut alors ajouter l'arc de v à u au chemin P pour construire un cycle de longueur impaire.
— Soit P est de longueur impaire, on peut alors remplacer l'arc v à u dans la séquence ci-dessus tout en préservant sa parité (impaire).

On pourrait répéter ces étapes pour tous les arcs qui sont dans le mauvais sens et ainsi construire un cycle dirigé impair $\mathcal{C}$.

Notons que l'ajout de P pourrait introduire des sommets non-distincts dans le cycle. Si c'est le cas, il suffit d'identifier les sous-cycles présents dans $\mathcal{C}$. Si un sous-cycle simple est de longueur paire, on peut le retirer sans affecter la parité de $\mathcal{C}$, s'il est de longueur impaire, on obtient un cycle qui répond aux demandes.

On a donc établi qu'à partir de C – un cycle non-dirigé impair dans une composante fortement connexe – on peut construire un cycle dirigé impair. Bien que cette construction ne soit pas de complexité linéaire, il nous suffit d'établir l'existence de C pour démontrer qu'il existe un cycle dirigé impair.

Pour ce faire, on cherche toutes les composantes fortement connexes et on établi pour chaque composante fortement connexe si son équivalent non-dirigé est biparti.

La recherche de composantes fortement connexes peut se faire en temps linéaire à l'aide de l'algorithme Kosaraju-Sharir vu au cours. Ensuite, pour chacune de ces composantes G' on peut vérifier si le graphe non-dirigé équivalent est biparti en temps linéaire en la taille de G' (cfr. séance 6). Puisque la somme des tailles de toutes les composantes fortement connexes est borné par la taille de G, cette deuxième partie se fait également en temps linéaire en la taille de G. □

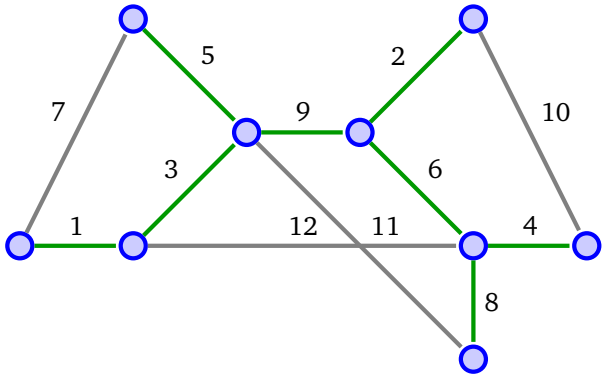
Séance 8 — Arbres couvrants

Exercice 8.1. Donner un algorithme qui, étant donné un graphe pondéré sur les arêtes, construit un arbre couvrant dont le poids maximum d'une arête est le plus petit possible (on appelle un tel arbre un MBST pour *Minimum Bottleneck Spanning Tree*).

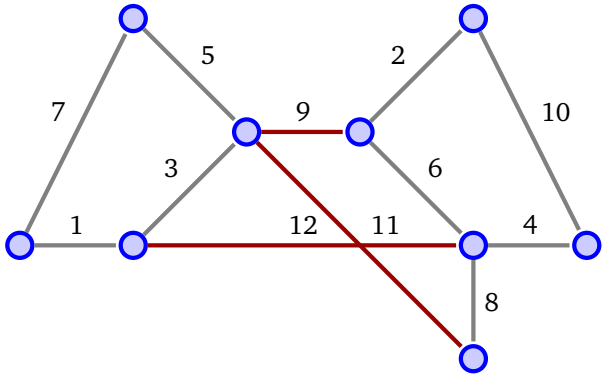
Résolution. Tout arbre couvrant de poids minimum est un arbre couvrant dont le poids maximum d'une arête est le plus petit possible. Nous allons avoir besoin de la proposition suivante :

Proposition 8.1. *Si T est un arbre couvrant d'un certain graphe connexe G pondéré sur les arêtes, alors pour toute arête e de T, e induit une coupe de G via T (notons-la $C_e(T)$ ou C_e si le contexte est clair) telle que $C_e \cap E(T_1) = \{e\}$.*

Démonstration. Cette arête e sépare deux composantes connexes de T. En effet, si le graphe $T - e$ est connexe, alors nous avons, par définition, l'existence d'un chemin P de $T - e$ joignant les deux sommets incidents à e. Dès lors $P + e$ est un cycle de T ce qui est impossible puisque T est un arbre. Dès lors e doit obligatoirement être la seule arête de C_e à appartenir à T. ■



(a)



(b)

FIGURE 31 – Illustration de la proposition ci-dessus : (a) Graphe avec un MST T (en vert). (b) Coupe (en rouge) contenant l'arête de poids maximum dans T (cette arête a un poids égal à 9). Cette coupe sépare les sommets en deux parties selon qu'ils se trouvent d'un côté ou de l'autre de l'arête de poids 9 dans T.

Montrons maintenant que tout MST est un MBST. Pour cela, prenons un graphe connexe quelconque G et deux arbres couvrants T_1 et T_2 tels que T_1 est un MST et T_2 est un MBST. En particulier, pour tout arbre couvrant T de G :

$$\max_{e \in T_2} w(e) \leq \max_{f \in E(T)} w(f).$$

Fixons e une arête de poids maximal de T_1 (qui n'est pas nécessairement unique). Par la proposition ci-dessus (puisqu'un MST est en particulier un arbre couvrant), nous savons que e est la seule arête de la coupe $C_e(T_1)$ à appartenir à T_1. Puisque C_e partitionne les sommets en deux, T_2 admet au moins une arête e' de C_e.

Cependant, par la propriété des coupes, nous savons que $\forall f \in C_e : w(e) \leq w(f)$. En particulier :

$$w_{\max}(T_1) = w(e) \leq w(e') \leq \max_{f \in E(T_2)} w(e) = w_{\max}(T_2).$$

Nous pouvons alors conclure que $w_{\max}(T_1) = w_{\max}(T_2)$, i.e. tout MST est en particulier un MBST.

□

Exercice 8.2. Donner un algorithme qui, étant donné un graphe pondéré sur les arêtes, construit un arbre couvrant dont le poids médian d'une arête est le plus petit possible.

Résolution. **Remarque : correction en cours de réécriture.** □

*** Exercice 8.3.** Donner un algorithme qui, étant donné un graphe pondéré sur les arêtes, détermine si l'arbre couvrant de poids minimum est unique.

Résolution. La procédure consiste à créer un arbre couvrant de poids minimum T à l'aide de l'algorithme de Kruskal, en choisissant au hasard la prochaine arête lorsqu'il y a plusieurs possibilités (c-à-d plusieurs arêtes de poids minimum ne faisant pas de cycle avec l'arbre en construction). Ensuite, on recommence en veillant à choisir, lorsqu'on a plusieurs possibilités, une arête n'appartenant pas à l'arbre T. Si une telle arête existe, cela signifie que l'arbre couvrant de poids minimum n'est pas unique. Si toutefois, lors du deuxième appel de l'algorithme, on n'a pas rencontré d'arête de poids minimum ne formant pas de cycle avec l'arbre en construction et n'appartenant pas à T, cela signifie que T est l'unique arbre couvrant de poids minimum.

Pour prouver que cette procédure est correcte, nous devons tout d'abord prouver qu'utiliser l'algorithme de Kruskal en choisissant au hasard l'arête parmi celles de poids minimum ne formant pas de cycle mène bien à un arbre couvrant de poids minimum. Pour cela, utilisons la proposition suivante :

Proposition 8.2. *Dans un graphe G pondéré sur les arêtes, pour toute coupe C de ce graphe, s'il existe plusieurs arêtes de poids minimum dans cette coupe, chacune d'elle appartient à au moins un arbre couvrant de poids minimum de ce graphe.*

Démonstration. Choisissons au hasard une arête e de poids minimum dans cette coupe. Prouvons qu'il existe un arbre couvrant de poids minimum contenant e. Considérons un arbre couvrant de poids minimum T ne contenant pas l'arête e et ajoutons celle-ci à cet arbre. Un cycle est nécessairement généré contenant au moins une autre arête dans la coupe C. Choisissons une de ces arêtes et nommons-la f. Remplaçons maintenant dans l'arbre T l'arête f par l'arête e pour obtenir l'arbre couvrant T'. Deux cas de figure se présentent :

— soit $w(f) > w(e)$,
— soit $w(f) = w(e)$.

Le premier cas n'est pas possible car le poids de T' serait inférieur à celui de T et ce dernier ne serait pas un arbre couvrant de poids minimum. Dans le deuxième cas, le poids de T' est égal à celui de T et donc T' est un arbre couvrant de poids minimum contenant l'arête e. ■ Il est facile maintenant de voir que l'algorithme de Kruskal proposé engendre bien un arbre couvrant de poids minimum car :

— L'arête choisie à chaque étape ne crée pas de cycle. Elle appartient donc bien à une coupe.
— L'arête choisie est l'une des arêtes de poids minimum de cette coupe.

Il est clair également que, lors du deuxième appel à l'algorithme, lorsqu'on veille à choisir une arête n'appartenant pas à l'arbre T, on obtient un arbre couvrant de poids minimum, puisqu'il s'agit d'une possibilité parmi d'autres. Il reste donc à prouver que tous les arbres couvrant de poids minimum peuvent être obtenus à l'aide de l'algorithme de Kruskal. En effet, dans ce cas, on peut dire que, lorsqu'on ne rencontre pas lors du deuxième appel d'arête n'appartenant pas à T parmi les arêtes possibles, l'arbre couvrant de poids minimum est bien unique. Pour cela, nous avons encore besoin de la proposition suivante :

Proposition 8.3. *Dans un graphe G pondéré sur les arêtes, pour toute coupe C et pour tout arbre couvrant de poids minimum T, ce dernier contient au moins une des arêtes de poids minimum de cette coupe.*

Démonstration. Démontrons cette proposition par l'absurde en supposant qu'elle soit fausse. Soit une coupe C et un arbre couvrant de poids minimum T de ce graphe. Soit e une des arêtes de poids minimum de cette coupe. Lorsqu'on ajoute cette arête à l'arbre T, on obtient un cycle contenant au moins une autre arête de cette coupe. Choisissons une telle arête et appelons la f. On sait que $w(f) = w(e)$ car, dans le cas contraire, T ne serait pas un arbre couvrant de poids minimum. Lorsqu'on remplace, dans l'arbre T, l'arête f par l'arête e, on obtient un arbre couvrant de poids minimum T' qui contient e, ce qui contredit notre hypothèse de départ. ■ Lors de chaque étape de l'algorithme de Kruskal, lorsqu'on considère toutes les arêtes de poids minimum ne formant pas de cycle avec l'arbre on construction, la dernière proposition nous permet de déduire que tout arbre couvrant de poids minimum contient au moins une de ces arêtes. Cela implique que tout arbre couvrant de poids minimum peut être construit à l'aide de l'algorithme de Kruskal et conclut notre démonstration.

□

Exercice 8.4. Coupe-cycles d'arêtes. Un ensemble d'arêtes est un *coupe-cycle* s'il contient au moins une arête de chacun des cycles du graphes. Si l'on enlève toutes les arêtes d'un coupe-cycle, le graphe résultant est acyclique (une forêt). Donner un algorithme efficace pour trouver, étant donné un graphe pondéré positivement sur les arêtes, un coupe-cycle d'arêtes de poids minimum.

Résolution. Supposons tout d'abord que le graphe donné $G = (V, E)$ est connexe (s'il ne l'est pas on peut répéter la procédure ci-dessous pour chaque composante connexe). On souhaite trouver un coupe-cycle C de poids minimum. De façon équivalente, on peut dire qu'on désire maximiser le poids de $E - C$. Puisque C est un coupe-cycle et que G est connexe, le graphe $G' = (V, E - C)$ est un arbre. Trouver un coupe-cycle de poids minimum est donc équivalent à trouver un arbre de poids maximum sur G. Ce dernier problème peut se résoudre en inversant le poids des arêtes et en appliquant l'algorithme de Kruskal. En résumé, on peut donc appliquer l'algorithme de Kruskal sur chaque composante connexe (avec poids inversés) et préserver comme coupe-cycle les arêtes qui sont présentes dans aucun des arbres couvrants. Trouver les composantes connexes se fait en temps proportionnel à $V + E$, appliquer l'algorithme de Kruskal se fait en temps proportionnel à $E \log E$, et déterminer l'ensemble d'arêtes absentes des arbres couvrants peut se faire en temps proportionnel à E. □

Exercice 8.5. Olympiades américaines d'informatique. Dans une ville, il y a n maisons, chacune nécessitant une arrivée d'eau. Le coût de construire un puits pour la maison `i` est `w[i]`. Le coût de construction d'un tuyau d'eau entre la maison `i` et la maison `j` est `c[i][j]`. Donner un algorithme qui calcule le coût minimum d'une solution d'approvisionnement de la ville en eau.

Résolution. Étant donné les n maisons, on construit un graphe contenant $n + 1$ sommets : un sommet par maison et un sommet supplémentaire représentant un puits. On ajoute ensuite une arête de poids $c[i][j]$ entre chaque pair de maisons i et j ainsi qu'une arête de poids $w[i]$ entre chaque maison i et le sommet puits. Pour déterminer la solution d'approvisionnement de coût minimal on calcule l'arbre couvrant de poids minimum

de ce graphe. Dans cet arbre, chaque maison est connectée soit (1) directement à un puits (si l'arbre contient une arête entre la maison et le sommet puits), ou (2) à un tuyau provenant d'une autre maison. Puisque l'abre est couvrant, au moins une des maisons sera connectée à un puits, et toutes les maisons sont connectées à une arrivée d'eau. De plus, puisque l'arbre couvrant est minimal, le coût du réseau est minimisé. □

Exercice 8.6. Étant donné un graphe dont les arêtes sont soit rouges, soit vertes, et un nombre k, donner un algorithme qui construit un arbre couvrant contenant exactement k arêtes rouges, ou certifie qu'il n'en existe pas.

Résolution. On établit en premier lieu les arbres couvrants contenant le plus et le moins d'arêtes rouges possibles. Pour trouver l'arbre couvrant contenant le moins d'arêtes rouges possibles, on assigne à chaque arête rouge un poids de 1 et à chaque arête verte un poids de 0, et on applique ensuite l'algorithme de Kruskal pour trouver l'arbre couvrant minimal T_{min}. Cet arbre minimisera le nombre d'arêtes rouges. De façon symétrique on peut obtenir l'arbre couvrant T_{max} contenant le plus d'arêtes rouges possibles. Notons k_{min} et k_{max} le nombre d'arêtes rouges dans l'arbre T_{min} et T_{max} respectivement. Si k est inférieur à k_{min} ou supérieur à k_{max} il suit que l'arbre désiré n'existe pas. Sinon, l'arbre désiré est soit T_{min}, soit T_{max}, soit il se trouve entre ces deux arbres.

Pour établir l'arbre T contenant $k_{min} < k < k_{max}$ arêtes rouges on part de $T = T_{min}$. Ensuite, et tant que le nombre d'arêtes rouges dans T est inférieur à k, on choisit une arête rouge dans T_{max} qui n'est pas dans T et on l'ajoute à T. Puisque T était un arbre cet ajout crée temporairement un cycle. Pour rétablir l'arbre on choisit une arête dans le cycle créé qui n'est pas dans T_{max} (il en existe toujours une puisque T_{max} est un arbre). L'arête retirée est soit verte, dans quel cas on a incrémenté le nombre d'arêtes rouges dans T, soit rouge, dans quel cas le nombre d'arêtes rouges reste inchangé. Ce deuxième cas ne peut avoir lieu qu'au plus k_{min} fois (quand aucune des arêtes rouges de T_{min} se trouve dans T_{max}), et, puisqu'il y a k_{max} arêtes dans T_{max} qu'on peut considérer pour ajout à T, il y aura au minimum $k_{max} - k_{min}$ échanges possibles qui incrémentent le nombre d'arêtes rouges dans T : suffisamment d'échanges pour atteindre toutes les valeurs de k entre k_{min} et k_{max}.

Trouver T_{min} et T_{max} se fait en $O(E \log E)$, le nombre d'arêtes à remplacer est borné par $k_{max} - k_{min} \leq E$, et chaque remplacement peut se faire en $O(V + E)$ (afin de déterminer le cycle créé et en supprimer une arête). Ces remplacements dominent la complexité totale : $O(E(V + E))$. □

Exercice 8.7. Algorithme de Boruvka. On se donne la méthode suivante pour construire un arbre couvrant de poids minimum : on ajoute des arêtes à une forêt, par phases. Initialement, la forêt est constituée des sommets isolés. À chaque phase, pour chaque arbre, on trouve une arête de poids minimum connectant cet arbre à un autre. On ajoute toutes ces arêtes dans la forêt, divisant donc le nombre d'arbres de la forêt (au moins) par deux. Donner les détails d'une implémentation efficace de cet algorithme et de sa complexité.

Résolution. Nous allons utiliser la structure `Union-Find` afin de pouvoir déterminer efficacement si deux sommets appartiennent au même arbre de la forêt.

Nous allons, à chaque phase de l'algorithme, construire un tableau dans lequel nous allons stocker, pour chaque sommet, quelle est l'arête de poids minimum liant ce sommet à un autre arbre de la forêt (si une telle arête existe) ; une fois ce tableau construit, nous allons ajouter toutes les arêtes qui lient effectivement deux composantes connexes différentes. Voici une implémentation possible :

```
1 public Queue<Edge> mstBoruvka(EdgeWeightedGraph G) {
2     UF uf = new UF(G.V());
3     Queue<Edge> mst = new Queue<Edge>();
4     while(mst.size() < G.V()-1) {            // O(log V)
```

```
        Edge[] minWeight = new Edge[G.V()];
        for(Edge e : G.edges()) {              // Theta(E)
            int v = e.either(), w = e.other(v);
            int i = uf.find(v), j = uf.find(w);
            if(i == j)
                continue;
            if(minWeight[i] == null || e.compareTo(minWeight[i]) < 0)
                minWeight[i] = e;
            if(minWeight[j] == null || e.compareTo(minWeight[j]) < 0)
                minWeight[j] = e;
        }
        for(int i = 0; i < G.V(); ++i) {       // Theta(V)
            Edge e = minWeight[i];
            if(e == null)
                continue;
            int v = e.either(), w = e.other(v);
            if(uf.find(v) == uf.find(w))
                continue;
            mst.enqueue(e);
            uf.union(v, w);
        }
    }
    return mst;
}
```

La boucle principale s'effectue $O(\log V)$ fois. En effet, notons C le nombre de composantes connexes (ou le nombre d'arbres dans la forêt) ; le nombre d'entrées de `minWeight` qui ne sont pas nulles est au moins C, pour un minimum de $\frac{C}{2}$ arêtes distinctes. Ajouter ces arêtes au MST en cours de construction va donc lier au moins $\frac{C}{2}$ arbres deux à deux, et donc diviser le nombre de composantes connexes restantes par (au moins) deux.

Le contenu de la boucle `while` principale demande un temps $\Theta(E+V) = \Theta(E)$ puisque toutes les arêtes sont visitées une première fois afin de déterminer les arêtes de poids minimum incidentes à chaque sommet, et puis les V arêtes retenues sont reparcourues en potentiellement ajoutées au MST en cours de construction, le tout en temps constant (car nous considérons les opérations sur `UF` comme se déroulant en temps constant).

Dès lors, le nombre total d'instructions est $O(E \log V)$.

□

Séance 9 — Plus courts chemins

Exercice 9.1. Prouver que pour tout plus court chemin entre deux sommets s et t passant successivement par deux autres sommets v et w, la portion du chemin comprise entre v et w est un plus court chemin entre v et w.

Résolution. Prouvons ceci par l'absurde. Supposons que ce ne soit pas le cas et qu'il existe un chemin entre les sommets v et w qui soit plus court que la portion du plus court chemin entre s et t comprise entre v et w. Définissons un autre chemin entre s et t qui soit égal au plus court chemin considéré précédemment à la différence près que la portion entre v et w est bien le plus court chemin entre ces deux sommets. Ce deuxième chemin est plus court que le premier chemin et ce dernier n'est donc pas un plus court chemin entre s et t. Cette contradiction prouve que notre hypothèse de départ était fausse. □

Exercice 9.2. Est-il vrai que si l'on ajoute un nombre c à tous les poids d'un graphe pondéré sur les arêtes, la solution du problème des plus courts chemins à une source ne change pas ?

Résolution. La réponse est *faux*. Un contre-exemple se trouve à la figure 32. □

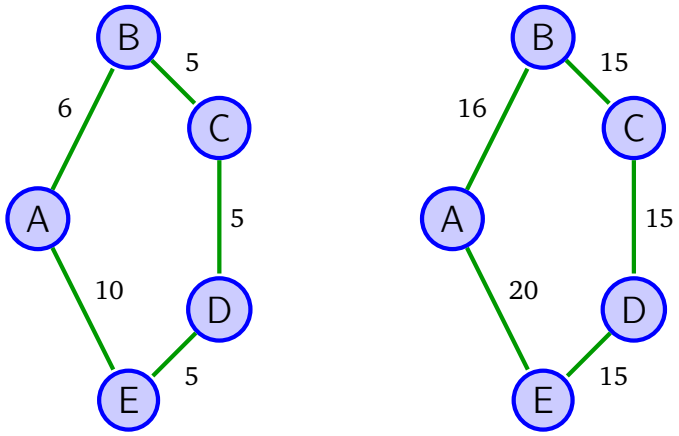


FIGURE 32 – Le graphe (b) est obtenu en ajoutant 10 à tous les poids du graphe (a). On voit que le plus court chemin entre les noeuds B et E n'est pas le même pour les deux graphes.

Exercice 9.3. On se donne un graphe *chemin*, dont tous les sommets sont de degré deux, sauf deux, qui sont de degré un. Les arêtes sont pondérées positivement. Donner un algorithme qui effectue un prétraitement du graphe en temps linéaire, et permet ensuite de retrouver les distances entre toute paire de sommets en temps constant.

Résolution. La solution consiste à choisir une des deux extrémités comme sommet de référence (ceci peut être fait en temps linéaire) et ensuite à parcourir le graphe à partir de ce sommet. De cette manière, on peut calculer les distances des autres sommets au sommet de référence en temps linéaire. La distance entre deux sommets est la valeur absolue de la différence de la distance au sommet de référence.

Listing 1 – Path.java

```
1 public class Path {
```

```
    private double[] distTo; // distances au sommet de référence
    private EdgeWeightedGraph G;

    public Path(EdgeWeightedGraph G) {
        this.G = G;
        distTo = new double[G.V()]; // distance au sommet de référence
        pretreatment();
    }

    /**
     * trouve une extrémité (au hasard) du graphe chemin
     * @return une extrémité du graphe
     */
    private int findExtremity() {
        boolean marked[] = new boolean[G.V()];
        int v = 0;
        marked[v] = true;
        Edge ed = null;
        int sum = 0;
        // vérifie si le sommet v est une extrémité (sum == 1)
        for (Edge e: G.adj(v)) {
            sum++;
            ed = e;
        }
        if (sum == 1) return v;
        int w = ed.other(v);
        marked[w] = true;
        while (true) {
            int next = -1;
            for (Edge edge: G.adj(w)) {
                int x = edge.other(w);
                if (marked[x])
                    continue;
                marked[x] = true;
                next = x;
                break;
            }
            if (next == -1) return w;
            w = next;
        }
    }

    /**
     * Calcule les distances au sommet de référence v (à  une extrémité) par les sommes
     * partielles des poids des arêtes
     * @param v: sommet de référence (à  une extrémité)
     */
    private void computeDistToRef(int v) {
        boolean marked[] = new boolean[G.V()];
        marked[v] = true;
        double dist = 0;
```

```
53          distTo[v] = dist;
54          while (true) {
55              int next = -1;
56              for (Edge e: G.adj(v)) {
57                  int w = e.other(v);
58                  if (marked[w])
59                      continue;
60                  marked[w] = true;
61                  next = w;
62                  dist += e.weight();
63                  distTo[w] = dist;
64              }
65              if (next == -1) break;
66              v = next;
67          }
68      }
69
70      /**
71       * procède aux prétraitement: choisit un sommet à une extrémité comme sommet de
             référence
72       * et calcule les distances des autres sommets à ce sommet de référence
73       */
74      private void pretreatment() {
75          computeDistToRef(findExtremity());
76      }
77
78      /**
79       * calcule la distance entre les sommets v et w en temps constant
80       * @param v: premier sommet
81       * @param w: deuxième sommet
82       * @return distance entre v et w
83       */
84      public double computeDist(int v, int w) {
85          return Math.abs(distTo[v] - distTo[w]);
86      }
87
88      /**
89       * test
90       */
91      public static void main(String[] args) {
92          int V = 10;
93          generateRandomPath(V);
94          Path p = new Path(G);
95          StdOut.println(p);
96          double dist = p.computeDist(0,1);
97          StdOut.println("distance entre les sommets 0 et 1: " + dist);
98      }
99  }
```

□

Exercice 9.4. Donner un algorithme de complexité linéaire pour le problème des plus courts chemins à une source dans un graphe dirigé *acyclique*.

Résolution. On peut facilement exploiter l'absence de cycle pour éviter l'utilisation d'une structure de file de priorité. En effet, nous avons vu qu'un graphe dirigé est acyclique si et seulement s'il possède un ordre topologique, tel que tout arc est compatible avec l'ordre. On peut trouver les plus courts chemins d'une source vers tous les autres sommets en considérant les sommets dans un ordre topologique.

Notons d'abord que dans l'algorithme de Dijkstra, l'opération de relaxation d'un arc est toujours effectuée sur toutes les arcs incidents partant d'un sommet donné. On définit naturellement l'opération de relaxation d'un *sommet* de cette façon.

```
private void relax(EdgeWeightedDigraph G, int v) {
  for (DirectedEdge e : G.adj(v))
    relax(e);
}
```

Un algorithme efficace de plus courts chemins à une source dans un graphe dirigé acyclique est le suivant : appeler la méthode `relax` sur tous les sommets du graphe dans un ordre topologique. On initialise préalablement le tableau `distTo[]` à 0 pour le sommet source `s` et à l'infini partout ailleurs.

Les conditions d'optimalité sont satisfaites, puisqu'au moment de traiter le sommet v, on a également traité tous les sommets le précédant dans l'ordre topologique, et donc tous les arcs de u vers v pour tout sommet u. L'algorithme est donc correct.

Un ordre topologique peut être obtenu via un parcours en profondeur en temps linéaire. Le coût total des relaxations des sommets est proportionnel au nombre E d'arcs. La complexité de l'algorithme est donc bien proportionnelle à $V + E$.

□

Exercice 9.5. Le triathlon des Flandres. On considère un graphe non dirigé pondéré sur les arêtes, et dont chaque arête est d'un des trois types suivant : voie d'eau, piste cyclable, ou piste de course. On souhaite connaître tous les plus courts chemins d'un sommet source s du graphe à tous les autres, avec la condition supplémentaire que les chemins doivent être constitués d'une suite d'arêtes de type voie d'eau, suivie d'une suite d'arêtes de type piste cyclable, suivies d'une suite d'arêtes de type piste de course. Chaque suite est constituée d'au moins une arête. Donner un algorithme efficace.

Résolution. La procédure consiste à utiliser une adaptation de l'algorithme de Dijkstra pour chaque partie nage/vélo/course du chemin, en veillant à ce que chaque partie contienne au moins une arête.

```
public class TriathlonSP {

   private IndexMinPQ<Double> pq;    // file prioritaire de sommets
   private double[] swimDistTo;      // swimDistTo[v] = distance du plus court chemin
       origin->v en nage
   private double[] bikeDistTo;      // bikeDistTo[v] = distance du plus court chemin
       origin->v en nage et vélo
   private double[] runDistTo;       // runDistTo[v] = distance du plus court chemin
       origin->v en nage, vélo et course
   private Edge[] swimEdgeTo;        // swimEdgeTo[v] = dernière arête sur plus court
       chemin origin->v en nage
```

```
8       private Edge[] bikeEdgeTo;        // bikeEdgeTo[v] = dernière arête sur plus court
            chemin origin->v en vélo
9       private Edge[] runEdgeTo;         // runEdgeTo[v] = dernière arête sur plus court
            chemin origin->v en course
10      private int origin;
11
12      /**
13       * Calcule les plus courts chemins dans le graphe G en nage, vélo et course à partir
             du sommet s vers tous les
14       * autres sommets du graphe
15       * @param G : graphe de type Triathlon
16       * @param s : sommet origine
17       */
18      public TriathlonSP(TriathlonGraph G, int s) {
19          pq = new IndexMinPQ<Double>(G.V());
20          origin = s;
21          swimSP(G);
22          bikeSP(G);
23          runSP(G);
24      }
25
26
27      /**
28       * Calcule les plus courts chemin en nage dans le graphe G à partir du sommet
             origine vers tous les autres
29       * sommets du graphe
30       * @param G: graphe de type triathlon
31       */
32      private void swimSP(TriathlonGraph G) {
33          swimDistTo = new double[G.V()];
34          swimEdgeTo = new Edge[G.V()];
35
36          for (int v = 0; v < G.V(); v++)
37              swimDistTo[v] = Double.POSITIVE_INFINITY;
38
39          // relâche premier sommet: nécessaire car il doit y avoir au moins une arête de
                nage dans le chemin
40          swimDistTo[origin] = 0;
41          for (Edge e : G.adj(origin)) {
42              if (TriathlonGraph.getType(e)==TriathlonGraph.EdgeType.swim) {
43                  relax(e, origin, swimDistTo, swimEdgeTo);
44              }
45          }
46
47          // nécessaire car il doit y avoir au moins une arête de nage dans le chemin
48          swimDistTo[origin] = Double.POSITIVE_INFINITY;
49
50          // cas où il y a une (ou plusieurs) arête de nage entre l'origine et l'origine
51          for (Edge e : G.adj(origin)) {
52              if (TriathlonGraph.getType(e)==TriathlonGraph.EdgeType.swim &&
                    e.other(origin)==origin) {
```

```
53                 if (swimDistTo[origin] > e.weight()) {
54                     swimDistTo[origin] = e.weight();
55                     swimEdgeTo[origin] = e;
56                 }
57             }
58         };
59 
60         // relâche sommets par ordre de distance à l'origine
61         while (!pq.isEmpty()) {
62             int v = pq.delMin();
63             for (Edge e : G.adj(v))
64                 if (TriathlonGraph.getType(e)==TriathlonGraph.EdgeType.swim){
65                     relax(e, v, swimDistTo, swimEdgeTo);
66                 }
67         }
68     }
69 
70     /**
71      * Calcule les plus courts chemin en nage et vélo dans le graphe G à partir du
            sommet origine vers tous les autres
72      * sommets du graphe
73      * @param G: graphe de type triathlon
74      */
75     private void bikeSP(TriathlonGraph G) {
76         bikeDistTo = new double[G.V()];
77         bikeEdgeTo = new Edge[G.V()];
78         bikeOrRun(G, TriathlonGraph.EdgeType.bike, swimDistTo, bikeDistTo, bikeEdgeTo);
79     }
80 
81     /**
82      * Calcule les plus courts chemin en nage, vélo et course dans le graphe G à partir
            du sommet origine vers tous les autres
83      * sommets du graphe
84      * @param G: graphe de type triathlon
85      */
86     private void runSP(TriathlonGraph G) {
87         runDistTo = new double[G.V()];
88         runEdgeTo = new Edge[G.V()];
89         bikeOrRun(G, TriathlonGraph.EdgeType.run, bikeDistTo, runDistTo, runEdgeTo);
90     }
91 
92 
93     /**
94      * Fonction utilisée par les fonctions bikeSP et runSP qui permet de calculer les
            plus courts chemins en nage & vélo
95      * et en nage & vélo & course respectivement.
96      * @param G : graphe de type triathlon
97      * @param type : soit bike soit run
98      * @param prevDistTo : si type == bike, prevDistTo = swimDistTo; si type == run,
            prevDistTo = bikeDistTo
99      * @param distTo: si type == bike, distTo = bikeDistTo; si type == run, distTo =
```

```
         runDistTo
100      * @param edgeTo: si type == bike, edgeTo = bikeEdgeTo; si type == run, edgeTo =
             runEdgeTo
101      */
102     private void bikeOrRun(TriathlonGraph G, TriathlonGraph.EdgeType type, double[]
             prevDistTo, double[] distTo,
103                            Edge[] edgeTo) {
104
105         if (G.V() >= 0) System.arraycopy(prevDistTo, 0, distTo, 0, G.V());
106
107         // relâche tous les sommets accessibles en nage à partir du sommet origine
108         for (int v = 0; v < G.V(); v++) {
109             if (prevDistTo[v] != Double.POSITIVE_INFINITY) {
110                 for (Edge e: G.adj(v)) {
111                     if (TriathlonGraph.getType(e) == type) {
112                         relax(e, v, distTo, edgeTo);
113                     }
114                 }
115             }
116         }
117
118         // s'assure que tous les chemins ont au moins une arête de type vélo
119         for (int v = 0; v < G.V(); v++) {
120             if (edgeTo[v] == null) distTo[v] = Double.POSITIVE_INFINITY;
121             if (prevDistTo[v] != Double.POSITIVE_INFINITY) { // sommets accessibles via
                    le précédent moyen de transport
122                 for (Edge e: G.adj(v)) { // cas où il y a une (ou plusieurs) arête du type
                        "type" du sommet v au sommet v
123                     if (TriathlonGraph.getType(e) == type && e.other(v) == v) {
124                         if (distTo[v] > prevDistTo[v] + e.weight()) {
125                             distTo[v] = prevDistTo[v] + e.weight();
126                             edgeTo[v] = e;
127                         }
128                     }
129                 }
130             }
131         }
132
133         // relâche les sommets par ordre de distance à l'origine
134         while (!pq.isEmpty()) {
135             int v = pq.delMin();
136             for (Edge e : G.adj(v))
137                 if (TriathlonGraph.getType(e)==type){
138                     relax(e, v, distTo, edgeTo);
139                 }
140         }
141
142     }
143
144
145     /**
```

```
146         * Relâche arête e et met pq à jour si nécessaire
147         * @param e : arête
148         * @param v : sommet précédent
149         * @param distTo: distances aux sommets
150         * @param edgeTo: arête contenant le sommet dans le plus court chemin
151         */
152        private void relax(Edge e, int v, double[] distTo, Edge[] edgeTo) {
153            int w = e.other(v);
154            if (distTo[w] > distTo[v] + e.weight()) {
155                distTo[w] = distTo[v] + e.weight();
156                edgeTo[w] = e;
157                if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
158                else                pq.insert(w, distTo[w]);
159            }
160        }
161
162        public double distTo(int v) {
163            return runDistTo[v];
164        }
165
166        public boolean hasPathTo(int v) {
167            return runDistTo[v] < Double.POSITIVE_INFINITY;
168        }
169
170        /**
171         * Retourne le plus court chemin de l'origine au sommet v en nage & vélo & course
172         * @param v: un des sommets du graphe
173         * @return le plus court chemin
174         */
175        public Iterable<Edge> pathTo(int v) {
176            if (!hasPathTo(v)) return null;
177            Stack<Edge> path = new Stack<Edge>();
178            int w = v;
179            for (Edge e = runEdgeTo[w]; e != null; ) {
180                path.push(e);
181                w = e.other(w);
182                if (runDistTo[w] > bikeDistTo[w]) break;
183                e = runEdgeTo[w];
184            }
185            for (Edge e = bikeEdgeTo[w]; e != null; ) {
186                path.push(e);
187                w = e.other(w);
188                if (bikeDistTo[w] > swimDistTo[w]) break;
189                e = bikeEdgeTo[w];
190            }
191            for (Edge e = swimEdgeTo[w]; e != null; ) {
192                path.push(e);
193                w = e.other(w);
194                e = swimEdgeTo[w];
195                if (w == origin) {
196                    break;
```

```
            }
        }
        return path;
    }

    /**
     * test
     * @param args
     */
    public static void main(String[] args) {
        TriathlonGraph G = new TriathlonGraph(5, 7); // 5 sommets et 7 arêtes
        StdOut.println(G);
        // calcule plus courts chemins à partir du sommet origine 0
        int s = 0;
        TriathlonSP sp = new TriathlonSP(G, s);
        // affiche les plus courts chemins
        for (int t = 0; t < G.V(); t++) {
            if (sp.hasPathTo(t)) {
                StdOut.printf("%d to %d (%.2f) ", s, t, sp.distTo(t));
                for (Edge e : sp.pathTo(t)) {
                    StdOut.print(e + "(" + TriathlonGraph.getType(e) + ") ");
                }
                StdOut.println();
            }
            else {
                StdOut.printf("%d to %d      no path\n", s, t);
            }
        }
    }
}
```

□

Exercice 9.6. Plus court chemin avec joker. Donner un algorithme de complexité proportionnelle à $E \log V$ qui, étant donné un graphe dirigé pondéré positivement sur les arêtes et deux sommets s et t, trouve le plus court chemin de s vers t, où on est autorisé à changer le poids d'une seule arête en 0.

Résolution. Si notre graphe de départ est le graphe $G = (\mathcal{V}, \mathcal{E})$ de sommets $\{v_1, \ldots, v_n\}$, construisons un graphe $G' = (\mathcal{V}', \mathcal{E}')$ défini comme suit :

- initialisons G' à G ;
- ajoutons les sommets $v'_1, \ldots, v'_n$ qui forment des *copies* des sommets initiaux ;
- pour chaque arête $e = (v_i, v_j) \in \mathcal{E}$, ajoutons une arête (v'_i, v'_j) (donc entre les sommets dupliqués) de poids $w(e)$ et ajoutons une arête (v_i, v'_j) de poids 0.

Si nous avons un sommet $s = v_i$ de départ et un sommet $t = v_j$ d'arrivée, nous pouvons appliquer l'algorithme de Dijkstra sur G' entre les sommets v_i et $t' = v'_j$. Montrons maintenant que cette solution donne bien une solution au problème posé : nous obtenons bien le plus court chemin entre s et t avec la possibilité de passer le poids d'(au moins) une arête à 0 car les poids des arêtes de G sont positifs. De plus, il ne peut y avoir qu'une unique arête dont le poids est passée à 0 car le graphe G est dupliqué sur les sommets v', et qu'une fois passé des sommets initiaux aux sommets dupliqués (donc par une arête (v_i, v'_j)), il est impossible de revenir dans le graphe initial (car il n'y a aucune arête (v'_i, v_j)). □

Séance 10 — Programmation dynamique

Exercice 10.1. Modifier la procédure de recherche de la longueur maximum d'une sous-séquence commune de façon à :

1. renvoyer non seulement la longueur mais également une telle sous-séquence ;
2. n'utiliser qu'un espace supplémentaire de taille linéaire.

Résolution. Nous allons partir de la solution vue au cours qui détermine la *longueur* de la plus longue sous-séquence entre deux séquences x et y, et y ajouter les informations nécessaires pour pouvoir reconstruire la sous-séquence en question.

Dans la solution initiale, nous appliquons la relation de récurrence suivante :

$$L_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ L_{i-1,j-1} + 1 & \text{si } x_i = y_j \\ \max\{L_{i-1,j}, L_{i\,j-1}\} & \text{sinon.} \end{cases}$$

L'algorithme de programmation dynamique remplit une matrice `L`, ligne par ligne en partant de la première sur base de la relation ci-dessus. Si, lors de ce remplissage, nous sauvons à chaque étape de quelle direction nous venons, alors nous pourrons suivre ces directions depuis le coin inférieur droit jusqu'à arriver à la première ligne ou la première colonne.

Définissons donc une classe `LCS` dans laquelle nous avons une énumération `Arrow` qui nous permet de savoir de quelle direction on vient. Cette classe a uniquement les séquences `x` et `y` en attribut.

```
class LCS {
    private enum Arrow {
        UPPER_LEFT,
        LEFT,
        UP
    }

    public char[] x, y;

    LCS(char[] x, char[] y) {
        this.x = x;
        this.y = y;
    }
```

La méthode `find` ci-dessous correspond à la solution vue en cours, mais nous ajoutons progressivement dans notre tableau `arrows` la direction depuis laquelle on arrive à la valeur de `L[i][j]` (i.e. `UP` si la valeur de `L[i][j]` est celle ce `L[i-1][j]`, `LEFT` si c'est celle de `L[i][j-1]`, et `UPPER_LEFT` sinon).

```
public char[] find() {
    int m = x.length;
    int n = y.length;
    int L[][] = new int[m+1][n+1];
    Arrow arrows[][] = new Arrow[m+1][n+1];
    for(int i = 0; i <= m; ++i) {
        for(int j = 0; j <= n; ++j) {
            if(i == 0 || j == 0) L[i][j] = 0;
```

```
            else if(x[i-1] == y[j-1]) {
                L[i][j] = L[i-1][j-1] + 1;
                arrows[i][j] = Arrow.UPPER_LEFT;
            } else if(L[i-1][j] > L[i][j-1]) {
                L[i][j] = L[i-1][j];
                arrows[i][j] = Arrow.UP;
            } else {
                L[i][j] = L[i][j-1];
                arrows[i][j] = Arrow.LEFT;
            }
        }
    }
    return reconstruct(arrows, L[m][n], m, n);
}
```

Il ne nous reste plus qu'à reconstruire la sous-séquence sur base de notre tableau `arrows` : nous partons du coin inférieur droit (`i=m` et `j=n`) et nous décrémentons soit `i`, soit `j`, soit les deux sur base de la valeur de `arrows[i][j]`. De plus, si `arrows[i][j] == UPPER_LEFT`, c'est que nous venons du cas `x[i-1] == y[j-1]`, et nous devons donc ajouter cette valeur à notre sous-séquence de retour.

Attention tout de même : nous construisons cette solution de la fin vers le début. Nous allons donc remplir notre tableau `subsequence` depuis la fin.

```
private char[] reconstruct(Arrow[][] arrows, int length, int m, int n) {
    int i = m, j = n;
    char[] subseq = new char[length];
    int idx = length;
    while(i > 0 && j > 0) {
        switch(arrows[i][j]) {
            case UP:
                --i; break;
            case LEFT:
                --j; break;
            default:
                subseq[--idx] = x[i-1]; // == y[j-1]
                --i; --j; break;
        }
    }
    return subseq;
}
```

Concernant le second point, afin de rester linéaire en espace, nous allons appliquer le principe de Hirschberg.

Il est important de remarquer que dans la relation de récurrence ci-dessus, lors du calcul de $L_{i,j}$, nous n'avons besoin que de la ligne précédente. En effet, puisque soit $L_{i,j} = L_{i-1,j-1}$, soit $L_{i,j} = \max\{L_{i-1,j}, L_{i,j-1}\}$, nous n'avons besoin que des entrées de la ligne i à gauche de l'entrée en cours, ou de la valeur juste au-dessus. En partant de cette observation, nous remarquons que dans notre implémentation, nous n'avons besoin que de stocker la dernière ligne. En particulier, nous pouvons implémenter la relation sur $L_{i,j}$ comme suit :

```
int getLcs(char[] x, char[] y) {
    int m = x.length, n = y.length;
    int[] L = new int[n+1];
    for(int i = 1; i <= m; ++i)
        for(int j = 1; j <= n; ++j)
            L[j] = (x[i-1] == y[j-1]) ? 1+L[j-1] : max(L[j-1], L[j]);
    return L[n];
}
```

Nous avons donc pu réduire la complexité en espace de $\Theta(mn)$ à $\Theta(n)$ (et même en particulier $\Theta(\min\{m,n\})$ par symétrie du problème).

Malheureusement, ce raisonnement ne nous permet pas directement d'appliquer le même traitement aux *flèches* utilisées pour reconstruire la sous-séquence.

Remarque : il est possible de reconstruire la sous-séquence de taille maximale en espace linéaire et en temps $O(mn)$ en appliquant des concepts de D&C, mais nous ne verrons pas cela ici. □

Exercice 10.2. Modifier la procédure de recherche du nombre minimum de multiplications dans un produit de matrices de façon à ce qu'elle renvoie un parenthésage optimal.

Résolution. En appliquant le même principe que dans l'exercice précédent, nous pouvons, tout en construisant notre tableau `m`, *retenir* de quel produit nous venons. Ainsi, après avoir calculé toutes les valeurs, nous pouvons simplement suivre ces flèches afin de savoir dans quel ordre les multiplications de matrices sont effectuées.

```
String parenthesization(int r[]) {
    int n = r.length-1;
    int[][] m = new int[n+1][n+1];
    int[][] ks = new int[n+1][n+1];
    for(int ell = 1; ell < n; ++ell) {
        for(int i = 1; i <= n-ell; ++i) {
            int j = i+ell;
            int min = Integer.MAX_VALUE;
            int best_k = 0;
            for(int k = i; k < j; ++k) {
                int v = m[i][k] + m[k+1][j] + r[i-1]*r[k]*r[j];
                if(v < min) {
                    min = v;
                    best_k = k;
                }
                m[i][j] = min;
                ks[i][j] = best_k;
            }
        }
    }
    return reconstruct(ks, 1, n);
}
```

Nous pouvons alors utiliser ce tableau `ks` comme suit : nous voulons connaître le parenthésage pour le produit de M_1 à M_n qui est donc $(M_{1:k} \cdot M_{k+1:n})$ pour k la valeur `ks[1][n]`. Nous pouvons ensuite construire

récursivement un `String` qui contiendra la séquence de produits :

```
String reconstruct(int[][] ks, int beg, int end) {
    if(beg == end)
        return String.valueOf(end);
    int k = ks[beg][end];
    return "(" + reconstruct(ks, beg, k) + "*" + reconstruct(ks, k+1, end) + ")";
}
```

□

Exercice 10.3. Donner un algorithme efficace pour le calcul de la longueur d'une plus longue sous-séquence commune à trois chaînes de symboles. Peut-on résoudre en temps polynomial le problème de plus longue sous-séquence commune à un nombre arbitraire k de chaînes ?

Résolution. Nous pouvons repartir la relation de récurrence pour deux séquences et l'étendre pour prendre en considération trois séquences x, y, z de longueur respective ℓ, m, n. Pour tous $i \leq \ell, j \leq m, k \leq n$:

$$L_{i,j,k} = \begin{cases} 0 & \text{si } 0 \in \{i,j,k\} \\ L_{i-1,j-1,k-1} + 1 & \text{si } x_i = y_j = z_k \\ \max\{L_{i,j,k-1}, L_{i,j-1,k}, L_{i-1,j,k}\} & \text{sinon.} \end{cases}$$

Nous pouvons aisément écrire un algorithme de programmation dynamique remplissant progressivement un tableau `L` par ordre croissant d'indice en $\Theta(\ell mn)$.

De manière plus générale, si nous avons k séquences $x^{(i)}$ (pour $1 \leq i \leq k$) de longueur respective n_i, alors nous définissons pour tout k-uple $(i_1, i_2, \ldots, i_k)$ tel que pour tout $1 \leq j \leq k$ nous avons $0 \leq i_j \leq n_k$, alors nous pouvons établir la relation de récurrence suivante :

$$L_{i_1,\ldots,i_k} = \begin{cases} 0 & \text{si } \exists j \in [\![1,k]\!] \text{ s.t. } i_j = 0 \\ 1 + L_{i_1-1,i_2-1,\ldots,i_k-1} & \text{si tous les } x^{(j)}_{i_j} \text{ sont égaux} \\ \max_{j \in [\![1,k]\!]} L_{i_1,\ldots,i_{j-1},i_j-1,i_{j+1},\ldots,i_k} & \text{sinon.} \end{cases}$$

La longueur de la plus longue sous-séquence commune entre les séquences $x^{(i)}$ est alors donnée par $L_{n_1,n_2,\ldots,n_k}$.

À nouveau, il est possible d'écrire un algorithme de programmation dynamique en $\Theta(\prod_{j=1}^{k} n_j)$:

```
int as_idx(int[] ns, int[] indices, int k) {
    int idx = indices[0];
    for(int j = 1; j < k; ++j) {
        idx *= (ns[j]+1);
        idx += indices[j];
    }
    return idx;
}

int get_max(int[] L, int[] ns, int[] indices, int k) {
    int max = 0;
    for(int i = 0; i < k; ++i) {
        indices[i]--;
        int idx = as_idx(ns, indices, k);
```

```
        if(L[idx] > max) max = L[idx];
        indices[i]++;
    }
    return max;
}

boolean contains_zero(int[] indices, int k) {
    for(int i = 0; i < k; ++i) if(indices[i] == 0) return true;
    return false;
}

boolean all_equal(char[][] sequences, int[] indices, int k) {
    char v = sequences[0][indices[0]-1];
    for(int i = 1; i < k; ++i) if(sequences[i][indices[i]-1] != v) return false;
    return true;
}

void increment(int[] indices, int[] ns, int k) {
    int idx = k-1;
    ++indices[idx];
    while(idx > 0 && indices[idx] > ns[idx]) {
        indices[idx] = 0;
        ++indices[--idx];
    }
}

int get_all_prev(int[] L, int[] ns, int[] indices, int k) {
    for(int i = 0; i < k; ++i) --indices[i];
    int ret = L[as_idx(ns, indices, k)];
    for(int i = 0; i < k; ++i) ++indices[i];
    return ret;
}

int solve(char[][] sequences, int k) {
    int[] ns = new int[k];
    int prod = 1;
    for(int i = 0; i < k; ++i) {
        ns[i] = sequences[i].length;
        prod *= (ns[i]+1);
    }
    int[] L = new int[prod];
    int[] indices = new int[k];
    while(indices[0] <= ns[0]) {
        int idx = as_idx(ns, indices, k);
        if(contains_zero(indices, k)) L[idx] = 0;
        else if(all_equal(sequences, indices, k))
            L[idx] = 1+get_all_prev(L, ns, indices, k);
        else
            L[idx] = get_max(L, ns, indices, k);
        increment(indices, ns, k);
    }
```

```
66      return L[prod-1];
67  }
```

Cet algorithme n'est donc pas polynomial en k mais est bien exponentiel en k, plus précisément en $\Omega(n^k)$ pour $n = \min_j n_j$. □

Exercice 10.4 (Plus courte super-séquence commune.)**.** On veut calculer la longueur minimum d'une chaîne de symboles qui contient deux chaînes A et B données en entrée comme sous-séquences. Montrer que ce problème se réduit à une recherche de plus court chemin sur un graphe pondéré acyclique.

Résolution. On peut assez aisément se rendre compte que pour x, y, deux séquences de longueur respective m et n, la longueur s de la plus petite super-séquence de x et y satisfait l'équation suivante : $m + n = s + \ell$, où ℓ est la longueur de la plus grande sous-séquence commune entre x et y.

En effet, en prenant z, une LCS entre x et y (de longueur ℓ), nous pouvons y ajouter $m - \ell$ symboles de x et $n - \ell$ symboles de y. Dès lors il faut au plus $\ell + (m - \ell) + (n - \ell)$ symboles pour une SCS, i.e. $s \leq m + n - \ell$. De plus, si nous prenons Z de longueur s une SCS de x et y et que nous retirons les $n - \ell$ symboles de y n'apparaissant pas dans z, alors il nous reste une séquence Z' de longueur $s - n + \ell$ qui contient au moins tous les caractères de x, i.e. $s - n + \ell \geq m$, ou encore $s \geq m + n - \ell$.

Nous en déduisons alors que $s = m + n - \ell$.

Intéressons-nous maintenant à la réduction de ce problème à un problème de plus court chemin sur un DAG pondéré.

Nous construisons un graphe G dont les sommets sont les paires $(i, j) \in [\![0, m]\!] \times [\![0, n]\!]$ (que nous pouvons donc disposer selon une grille) et nous allons créer une arête entre les sommets (i, j) et (i', j'), pour $i' \geq i$ et $j' \geq j$ lorsque :
- $i' - i \leq 1$ et $j' - j \leq 1$ (sont voisins dans la grille, potentiellement en diagonale) ;
- $i' - i + j' - j \gneq 0$ (sont deux sommets différents).

Soit e une arête. Nous la pondérons comme suit :
- si e joint deux sommets en ligne ou en colonne, alors $w(e) = 1$;
- sinon (e joint deux sommets en diagonale), alors $w(e) = 0$ si $x_{i'} = y_{j'}$ et $w(e) = 2$ sinon.

En recherchant un plus court chemin entre les sommets $(0, 0)$ et (m, n), nous déterminons un chemin dans lequel nous allons maximiser les arêtes diagonales de poids nul, et donc maximiser le nombre de symboles en commun. Ces arêtes diagonales de poids nul qui sont prises dans le chemin forment une LCS entre x et y. Dès lors, par construction, le nombre d'arêtes du chemin détermine la longueur d'une SCS de x et y. De plus, ce nombre d'arêtes vaut $m + n - d$ où d est le nombre d'arêtes diagonales prises. Or d comme dit ci-dessus, d est la longueur de la LCS entre x et y, et nous retrouvons bien l'égalité démontrée au début de cet exercice.

Remarque : Nous pouvons également ne pas définir les arêtes diagonales de poids 2 dans la construction précédente car elles ne servent à rien. En faisant cela, nous pouvons retirer les poids sur les arêtes. □

Exercice 10.5. Étant donné une matrice carrée A d'entiers positifs et de taille $n \times n$, on considère les suites serpentines de A : des suites d'éléments de A tels que chaque élément est soit à droite, soit en-dessous de l'élément précédent dans la suite, et est tel que la différence avec l'élément précédent est 1 ou -1. Donner un algorithme efficace pour trouver la longueur maximum d'une suite serpentine dans la matrice A donnée en entrée.

Résolution. Construisons une table `L` de taille $n \times n$ initialisée à 1. L'entrée `L[i][j]` nous donne la longueur de la plus longue suite serpentine finissant en position (i, j). Puisque chaque élément d'une telle séquence doit se trouver à droite ou en dessous de l'élément précédent, nous allons pouvoir itérer sur `i` et `j` et regarder l'élément au dessus et l'élément à gauche :

```java
int abs(int x) { return x>0 ? x : -x; }
int max(int x, int y) { return x>y ? x : y; }

int snakeSequence(int[][] A, int n) {
    int longest = 0;
    int[][] L = new int[n][n];
    for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j)
            L[i][j] = 1; // il existe une séquence de longueur 1
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < n; ++j) {
            if(i > 0 && abs(A[i][j] - A[i-1][j]) == 1)
                L[i][j] = max(L[i][j], 1 + L[i-1][j]);
            if(j > 0 && abs(A[i][j] - A[i][j-1]) == 1)
                L[i][j] = max(L[i][j], 1 + L[i][j-1]);
            longest = max(longest, L[i][j]);
        }
    }
    return longest;
}
```

Il faut également, lors de chaque tour de boucle, vérifier si la séquence créée à cette étape est plus longue que la plus longue trouvée jusqu'à présent. □

Exercice 10.6. Étant donné un nombre k et une collection de n nombres $x_1, x_2, \ldots, x_n$, donner un algorithme de complexité $O(nk)$ qui décide s'il existe un sous-ensemble des n nombres dont la somme est égale à k. Cet algorithme est-il polynomial ?

Résolution.

```java
boolean exists(int[] x, int k) {
    int n = x.length;
    boolean[][] T = new boolean[n+1][k+1];
    for(int i = 0; i <= n; ++i)
        T[i][0] = true;
    for(int i = 1; i <= n; ++i) {
        for(int ell = 1; ell <= k; ++ell) {
            int y = x[i-1];
            T[i][ell] |= T[i-1][ell];
            if(y <= ell)
                T[i][ell] |= T[i-1][ell-y];
        }
    }
    return T[n][k];
}
```

Cet algorithme n'est pas polynomial mais est pseudo-polynomial puisqu'il est linéaire en la *valeur* n (mais pas en la taille de son encodage). □