TD
Fondements de l'Informatique
IN 100

2005-06

1 Base

Exercice 1.1. Écrire un algorithme qui échange le contenu de deux variables :

1. En utilisant une troisième variable temporaire.
2. En n'utilisant aucune autre variable.

Exercice 1.2. Écrire un algorithme qui prend en argument trois entiers `a`, `b` et `c` et qui met dans `c` le maximum de la valeur de `a` et de la valeur de `b`.

Exercice 1.3. Écrire un algorithme qui prend en argument trois entiers `a`, `b` et `c` et qui renvoie vrai dans une variables booléenne si $a^2 + b^2 = c^2$ et faux sinon.

Exercice 1.4. Donner les tables de vérité du `ET`, du `OU` et du `NON` logique. Dans la suite, pour tous booléens a et b :

- a `ET` b sera noté $a \wedge b$
- a `OU` b sera noté $a \vee b$
- `NON`(a) sera noté $\overline{a}$

1. Montrer que :
$$\overline{(a \wedge b)} = (\overline{a} \vee \overline{b})$$
2. Montrer que :
$$\overline{(a \vee b)} = (\overline{a} \wedge \overline{b})$$
3. Expliquer la table de vérité de l'implication notée $\Longrightarrow$:

a	b	$a \Longrightarrow b$
F	F	V
F	V	V
V	F	F
V	V	V

4. Montrer que :
$$(a \Rightarrow b) = (\overline{a} \vee b)$$
5. Donner la table de vérité de l'équivalence notée $\Longleftrightarrow$.
6. Exprimer l'équivalence avec les opérateurs `ET` et `NON`.
7. Donner la table de vérité d'un opérateur $\diamond$ tel que :
   - $(a \diamond a) = \overline{a}$
   - $(a \diamond b) = (\overline{a} \wedge \overline{b})$

   Exprimer tous les opérateurs logiques vu ci-dessus uniquement avec $\diamond$.
8. On considère la table de vérité ci-dessous :

a	b	ab
F	F	c_{FF}
F	V	c_{FV}
V	F	c_{VF}
V	V	c_{VV}

   où c_{FF}, c_{FV}, c_{VF}, c_{VV} sont des booléens. Montrer que quelquesoientt les valeurs que prennent ces quatre booléens, la table de vérité ci-dessus peux s'exprimer uniquement avec des `OU`, `ET` et `NON`.

Exercice 1.5.

1. Écrire un algorithme qui prend en argument trois réels `a`, `b` et `c` et qui calcule les racines du polynôme $ax^2 + bx + c$.
2. Ajouter à l'algorithme une vérification du résulat.
3. Quel problème d'implémentation pose cette vérification ?

Exercice 1.6. On considère la suite :

$$\begin{cases} u_0 = 0 \\ u_n = 2u_{n-1} + 1, \quad \forall n \geq 1 \end{cases}$$

Écrire un algorithme qui prend en entrée n et qui renvoie en sortie u_n.

Exercice 1.7. On considère la suite de Fibonacci :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2}, \quad \forall n \geq 2 \end{cases}$$

Écrire un algorithme qui prend en entrée n et qui renvoie en sortie u_n.

Exercice 1.8. Soit l'algorithme suivant :

```
Algorithme algo_1 (N : Entier)
Parametres d'Entrée
    N
Parametres locales
    i : Entier

Début_0
i <- 1
Tant Que (i <= N) Faire
     Début_1
     Si (est_pair(i))
        Alors Début_2
              Afficher(i)
              Fin_2
     i <- i+1
     Fin_1
Fin_0
```

1. Que fait cet algorithme.
2. Que se passe-t-il si $N \leq 0$.

Exercice 1.9. Soit l'algorithme suivant :

```
Algorithme algo_2 (N : Entier, S : Entier)
Parametres d'Entrée
    N
Parametres de Sortie
    S
Variables locales
    i : Entier
Début_0
S <- 0
i <- 1
Tant Que (i <= N) Faire
     Début_1
     S <- S+i
     i <- i+1
     Fin_1
Fin_0
```

1. Que fait cet algorithme.

2. Que se passe-t-il si $N \leq 0$.

Exercice 1.10. Soit l'algorithme suivant :

```
Algorithme algo_2 (N : Entier, cpt : Entier)
Parametres d'Entrée
    N
Parametres de Sortie
cpt {Compteur}
Début_0
i <- 1
cpt <- 0
Tant Que (i <= N) Faire
     Début_1
     j <- 0
     Tant que (i+j <= N) Faire
          Début_2
          cpt <- cpt+1
          j <- j+1
          Fin_2
     i <- i+1
     Fin_1
Fin_0
```

1. Combien vaut `cpt` à la fin de l'algorithme.
2. Programmer cet algorithme.

2 Tableaux

Exercice 2.1. Parmi les algorithmes ci-dessous, lequel fait ce qu'il dit faire, pour les autres dire ce qu'il font réellement.

```
Algorithme cherche_1 (N : Entier, T : Tableau d'Entier,
                      val : entier,
                      est_present : Booleen)
Description
    Si la valeur val est dans le tabelau T
    alors est_present contiendra Vrai,
    sinon est_present contiendra Faux.
```

```
Parametres d'Entrée
    N {La taille du Tableau}
    T {Le tabeau de N entiers}
    val {L'entier que l'on cherche dans le tableau}
Parametres de Sortie
    est_present : {Vrai si val appartient a T, Faux sinon}
Parametres locales
    i : Entier

Début_0
est_present <- Faux
i <- 1
Tant Que (i <= N) Faire
     Début_1
     Si (T[i] = val)
        Alors Début_2
              est_present = Vrai
              Fin_2
     i <- i+1
     Fin_1
Fin_0
```

```
Algorithme cherche_2 (N : Entier, T : Tableau d'Entier,
                      val : entier,
                      est_present : Booleen)
Description
    Si la valeur val est dans le tabelau T
    alors est_present contiendra Vrai,
    sinon est_present contiendra Faux.
Parametres d'Entrée
    N {La taille du Tableau}
    T {Le tabeau de N entiers}
    val {L'entier que l'on cherche dans le tableau}
Parametres de Sortie
    est_present : {Vrai si val appartient a T, Faux sinon}
Parametres locales
    i : Entier

Début_0
i <- 1
Tant Que (i <= N) Faire
```



```
     Début_1
     Si (T[i] = val)
        Alors Début_2
              est_present = Vrai
              Fin_2
        Sinon Début_3
              est_present = Faux
              Fin_3
     i <- i+1
     Fin_1
Fin_0
```

```
Algorithme cherche_3 (N : Entier, T : Tableau d'Entier,
                      val : entier,
                      est_present : Booleen)
Description
    Si la valeur val est dans le tabelau T
    alors est_present contiendra Vrai,
    sinon est_present contiendra Faux.
Parametres d'Entrée
    N {La taille du Tableau}
    T {Le tabeau de N entiers}
    val {L'entier que l'on cherche dans le tableau}
Parametres de Sortie
    est_present : {Vrai si val appartient a T, Faux sinon}
Parametres locales
    i : Entier

Début_0
i <- 1
Tant Que ((i <= N) ET (T[i] != val)) Faire
     Début_1
     i <- i+1
     Fin_1

Si (i <= N)
   Alors Debut_2
         est_present = Vrai
         Fin_2
   Sinon Debut_3
         est_present = Faux
```

```
        Fin_3
Fin_0
```

Exercice 2.2. Soit T un tableau de N entiers et x une valeur entière. Écrire un algorithme qui renvoie dans un argement en sortie l'indice de la case du tableau qui contient la première occurence de la valeur x. Cet argument ens ortie vaudra `-1` si la valeur x ne se trouve pas dans le tableau. Même question avec la dernière occurence.

Exercice 2.3. Soit T un tableau de N entiers et x une valeur entière. Écrire un algorithme qui renvoie dans un argement en sortie la valeur immédiatement supérieure à x. Que fait votre algorithme si x est la plus grande valeur du tableau.

Exercice 2.4. Soit T un tableau de N entiers. Écrire un algorithme qui renvoie en argument(s) de sortie
- la plus petite valeur stockée dans le tableau.
- la plus grande valeur stockée dans le tableau.
- la plus petite et la plus grande

Exercice 2.5. Soit T un tableau de N entiers. Écrire un algorithme qui renvoie dans un argement en sortie la somme des toutes les valeurs stockées dans le tableau.

Exercice 2.6.

```
Algorithme Mystere (N : Entier, T : Tableau d'Entier,
                    nb_etapes : Entier)
Description
    C'est la question
Parametres d'Entrée
    N {La taille du Tableau}
    T {Le tabeau de N entiers}
Parametres de Sortie
    nb_etapes {Le nombre d'étapes de l'algorithme}
Parametres locales
    i : Entier

Début_0
i <- 1
nb_etapes <- 0
```



```
Tant Que (i <= N) Faire
     Début_1
     i <- T[i]
     nb_etapes <- nb_etapes + 1
     Fin_1
Fin_0
```

1. Quelle sont la plus petite et la plus grande valeur que `nb_etapes` peut contenir à la fin de l'algorithme? Donner des exemples.
2. Donner un exemple où l'algorithme ne s'arrête jamais.

Exercice 2.7. On suppose que l'instruction :
`affiche(x)`
affiche la valeur contenue dans la variable `x`.

Écrire un algorithme qui affiche les valeurs contenue dans un tableau de `N` entiers de la case `1` à la case `N`.

Même question, en affichant de `N` à `1`.

Exercice 2.8. Soit un tableau T de N entier tel que :

$$\forall i, j \in 1..N, \quad i < j, \quad T[i] \leq T[j]$$

Dans ce cas, on dit que le tableau est trié.

Donner un algorithme qui recherche si une valeur x est présente dans le tableau.

Si on supposes que $\forall i, j \in 1..N$, $i \neq j$, $T[i] \neq T[j]$. Si onrecherche une valeur x qui est dans le tableau, en combien d'"étapes" trouve-t-on cette valeur :

1. dans le meilleur des cas,
2. dans le pire des cas,
3. en moyenne.

Même question, si x n'est pas dans le tableau.

3 Complexité

Dans les exercices suivants, lorsque des calculs de complexité seront demandés, il faudra avant tout définir quelles sont les opérations que l'on considère (il s'agira en général d'opérations arithmétiques sur les entiers et/ou de comparaisons).

Exercice 3.1. Ranger les fonctions suivantes par ordre de grandeur croissant :

$$n, \sqrt{n}, \log n, \log\log n, 2^{3^n}, n/\log n, \sqrt{n}.\log^2 n,$$
$$(3/2)^n, n^{10}, 10^n, \log^2 n, e^{n^2}, 1/3^n$$

Exercice 3.2. Evaluer la complexité de la fonction suivante :

```
Fonction mystere (N : Entier)
Entrée
    N : Entier
Sortie
Local
    i, j, k : Entier
Début
Pour i de 1 à N Faire
     Pour j de 1 à N Faire
          Pour k de 1 à N Faire
               Si (i<=j ET j<=k) Alors Action 4
Fin
```

Exercice 3.3.

- On considère un problème que l'on sait résoudre en temps n^2. Supposons qu'on sache le diviser en temps n^α en deux sous-problèmes de taille $n/2$. Pour quelles valeurs de α est-il asymptotiquement intéressant de diviser le problème en sous-problèmes ?
- On considère un problème que l'on sait résoudre en temps n^3. Supposons qu'on sache le diviser en temps 1 en β sous-problèmes de taille $n/2$. Pour quelles valeurs de β est-il asymptotiquement intéressant de diviser le problème en sous-problèmes ?

Exercice 3.4. Ecrire l'algorithme de somme de deux matrices carrées $n \times n$, puis en évaluer la complexité. Faire de même pour l'algorithme naïf de produit de deux matrices carrées $n \times n$.

Exercice 3.5. Tout polynôme de degré inférieur à n peut être représenté par le tableau `P[0] ... P[n]` de ses coefficients. Par commodité, on supposera que $n = 2^p - 1$.

- Ecrire un algorithme de calcul du produit de deux polynômes de degré n. Quelle est sa complexité ?

- Soit A et B deux polynômes de degré $2^p - 1$. On peut écrire de façon unique $A = A_1 \times X^{2^{p-1}} + A_2$ et $B = B_1 \times X^{2^{p-1}} + B_2$, où A_1, A_2, B_1, B_2 désignent des polynômes de degré $2^{p-1} - 1$. Exprimer AB en fonction de A_1, A_2, B_1, B_2. Montrer qu'on peut calculer ce produit en effectuant seulement trois multiplications de polynômes de degré $2^{p-1} - 1$. En conclure qu'on peut abaisser la complexité du calcul du produit de deux polynômes.

Exercice 3.6. Ecrire un algorithme qui teste la présence d'un entier dans le tableau `T`. En supposant que l'élément cherché à une probabilité p de se trouver dans le tableau, et que s'il s'y trouve sa position dans le tableau suit une loi uniforme, donner la complexité en moyenne de cet algorithme.

Exercice 3.7. Ecrire un algorithme (naïf) de recherche du plus grand élément d'un tableau. Quelle est sa complexité en terme d'affectations dans le meilleur, le pire cas et en moyenne ? Et en terme de comparaisons ? On cherche maintenant à déterminer, simultanément, le plus petit et le plus grand élément du tableau `T`. Donner un algorithme naïf qui résout ce problème. Quelle est sa complexité en nombre de comparaisons ? L'idée pour améliorer l'algorithme consiste à considérer les éléments du tableau non pas un par un, mais par paires. Décrire l'algorithme et donner sa complexité.

Exercice 3.8. On s'intéresse ici à des tableaux pouvant contenir plusieurs fois le même élément. On dit qu'un élément est majoritaire dans `T` s'il apparaît strictement plus de $n/2$ fois dans le tableau. Par commodité, on supposera ici que n est une puissance de 2.

- Ecrire un algorithme qui calcule le nombre d'occurrences d'un élément donné dans le tableau.
- En déduire un algorithme pour tester si le tableau possède un élément majoritaire.
- Déterminer un autre algorithme, récursif, basé sur le découpage de `T` en deux tableaux de même taille.

4 Dénombrement

Exercice 4.1. De combien de manières diffrentes peut-on remplir un tableau de taille n en utilisant des entiers compris entre 0 et $m - 1$? Qu'en est-il si on impose en plus que le même entier ne peut pas apparaitre plus d'une fois dans le tableau ?

Exercice 4.2. On considère un ensemble de n entiers distincts. De combien de manières différentes peut-on représenter cet ensemble au moyen d'une liste ?

Exercice 4.3. On considère un nombre entier x et on définit par récurrence une suite $(u_n)_{n\geq 0}$ de la manière suivante :

$$\begin{array}{rcl} u_0 & = & 1 \\ u_n & = & \begin{cases} (u_p)^2 & \text{si } n = 2p \text{ et } p > 0 \\ x \times (u_p)^2 & \text{si } n = 2p+1 \end{cases} \end{array}$$

Montrer par récurrence que $u_n = x^n$ quel que soit $n \geq 0$. En déduire un programme récursif calculant x^n à partir de x et n passés en paramètres. Quelle est sa complexité ?

Exercice 4.4. Le but du jeu est le suivant : on part d'une configuration où n disques sont empilés par ordre décroissant de taille sur un emplacement a et il reste deux emplacements libres b et c et on cherche à transférer un par un tous les disques vers l'emplacement c sans qu'à aucun moment il y ait un disque plus grand empilé sur un disque plus petit. On demande d'écrire un programme *récursif* prenant en entrée le nombre de disques et indiquant les manipulations a effectuer pour résoudre le problème. Calculer sa complexité.

Exercice 4.5. On considère les définitions suivantes :

```
typedef struct cell {
  int x;
  struct cell* suiv;
} Cell;

typedef Cell* list;
```

Ecrire des fonctions *récursives* permettant d'insérer, de supprimer, de tester la présence d'un élément dans de telles listes (on utilisera une représentation des listes faisant usage d'une première cellule "fictive").

Exercice 4.6. Rappeler le principe du tri fusion. Donner la formule de récurrence permettant de calculer sa complexité et la résoudre.

Exercice 4.7. La suite de Fibonacci est définie par la formule de récurrence suivante :

$$\begin{array}{l} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \text{ pour } n \geq 2 \end{array}$$

Ecrire une fonction permettant de calculer f_n à partir de l'entier n fourni en paramètre. On note T la complexité de cette fonction. Montrer que pour $n \geq 2$ on a :

$$2T(n-2) + 1 \leq T(n) \leq 2T(n-1) + 1$$

En déduire qu'il existe des constantes c_1 et c_2 telles que :

$$c_1^n = O(T(n)) \text{ et } T(n) = O(c_2^n)$$

Qu'en déduit-on sur la "classe de complexité" de cet algorithme ? Peut-on trouver plus efficace ?

Exercice 4.8. Ecrire une fonction *récursive* permettant de trouver la valeur du plus grand élément d'un tableau. Comment modifier le programme pour qu'il retourne la position du plus grand élément à l'intérieur de tableau (au lieu de sa valeur) ?

5 Algorithmes de tri

La complexité des algorithmes de tri sera calculée en fonction du nombre de comparaisons et/ou d'échanges.

Exercice 5.1. Considérons tout d'abord le tableau suivant :

$$T = [11, 66, 51, 83, 6, 18, 100, 3, 38, 27]$$

Pour $i \in \{0, \ldots, 9\}$, on pose :

$$C[i] = \text{card}\{j \in \{0, \ldots, 9\} \mid T[j] < T[i]\}$$

Caculer le tableau C. Comment l'utiliser pour trier le tableau T ? En déduire un algorithme de tri. Quelle est sa complexité ?

Exercice 5.2. On veut réaliser un programme permettant de trier un tableau par insertion. Pour cela, on demande de :

- écrire un programme prenant en entrée un tableau trié T de taille k et un entier x et renvoyant la position à laquelle il faut insérer x dans ce tableau,
- écrire un programme prenant en entrée un tableau à $k+1$ cases, en entier x et en entier $i \in \{0, \ldots, k-1\}$ et insère x à la position i dans T (le contenu de la dernière case de T peut être perdu),
- écrire un programme récursif de tri utilisant les deux programmes précédents.

Quelle est la complexité (au pire) de ce programme en nombre d'affectations ? En nombre de comparaisons ? Proposer une méthode pour améliorer la complexité en nombre de comparaisons.

Exercice 5.3. Programmer récursivement l'algorithme de tri par sélection d'un tableau, en plaçant à chaque fois le plus grand élément apparaissant dans le tableau à la fin. Ecrire un programme de tri par selection sans utiliser la récursivité. Calculer (en fonction de la taille du tableau) le nombre maximum d'échanges et de comparaisons effectués par le tri par sélection.

Exercice 5.4. Décrire l'algorithme de tri bulle et écrire le programme correspondant. Quelle est sa complexité au pire, en nombre de comparaisons et en nombre d'échanges ? On veut maintenant étudier la complexité en moyenne de cet algorithme. Pour cela, on ne va considérer que des tableaux T de taille n et qui contiennent tous les entiers entre 0 et $n-1$ (il s'agit donc des tableaux qui sont obtenus en mélangeant le tableau trié $[0, \ldots, n-1]$). Pour un tel tableau on note :

$$T = T[0], \ldots, T[n-1]$$
$$I(T) = \text{card}\{(i,j) \mid 0 \leq i < j < n \text{ et } T[i] > T[j]\}$$
$$\bar{T} = (n-1-T[0]), \ldots, (n-1-T[n-1])$$

On dit que $I(T)$ est le nombre d'inversions de T et que $\bar{T}$ est le tableau complémentaire de T. Comparer $I(T)$ et $I(\bar{T})$. Utiliser $I(T)$ pour calculer le nombre d'échanges réalisés lorsque le tri bulle est appliqué à T. En déduire la complexité moyenne du tri bulle en nombre d'échanges (la complexité moyenne en nombre de comparaisons est nettement plus difficile à établir).

Exercice 5.5. Décrire l'algorithme de partitionnement du tri rapide. Cet algorithme prend en entrée un tableau T de taille $n >= 1$ et un pivot p apparaissant dans T. Il retourne un indice k entre 0 et $n-1$ tel que :

- $k - 1 \geq 0$ et $k \leq n-1$,
- si $0 \leq i < k$, $T[i] \leq p$,
- si $k \leq i < n$, $T[i] \geq p$.

Expliquer de quelle manière chacun de ces point intervient dans la construction de l'algorithme de tri rapide. Programmer l'algorithme de partitionnement et détailler son fonctionnement sur le tableau suivant (le pivot est l'élément contenu dans la première case) :

$$[22, 36, 6, 79, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 47]$$

Exercice 5.6. En s'inspirant de l'algorithme de partitionnement du tri rapide, décrire un algorithme permettant de trier en une seule passe un tableau ne contenant que des 0 et des 1.

Exercice 5.7. Décrire l'algorithme de tri rapide en insistant bien sur les différentes composantes qui interviennent. Donner une implémentation.

Exercice 5.8.

- Appliquer l'algorithme de tri rapide au tableau suivant :

$$[22, 36, 6, 79, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 47]$$

- Effectuer un tri rapide sur les tableaux suivants :

$$[1, 2, 3, 4, 5, 6, 7]$$
$$[4, 6, 5, 2, 7, 1, 3]$$
$$[7, 1, 6, 2, 5, 3, 4]$$
$$[5, 3, 7, 2, 4, 1, 6]$$

  Que vous suggèrent ces quelques exemples ?
- Comment peut-on envisager de modifier la méthode de tri rapide pour obtenir (rapidement) le k-ième élément d'un tableau ? Quelle est la complexité au pire et en moyenne de cet algorithme ? Qu'en est-il si l'agorithme de choix du pivot garantit que la taille du sous-tableau de l'appel récursif est au plus αn, avec $\alpha < 1$?

Exercice 5.9. On veut construire un programme de tri qui ne modifie pas le tableau T mais produit un nouveau tableau S tel que le tableau :

$$T[S[1]], \ldots, T[S[n]]$$

soit le tableau obtenu en triant T. Construire un tel programme, en s'inspirant par exemple du tri bulle. Comment, à partir de ce programme, produire le tableau trié ?

Exercice 5.10. On cherche à réaliser un programme de tri qui ne modifie pas le tableau T passé en paramètre mais produit un nouveau tableau S tel que pour $0 \leq i < n$, $S[i]$ contient la nouvelle position de $T[i]$. Réaliser un tel programme de tri, en s'inspirant par exemple du tri bulle. Comment, à partir de ce programme, produire le tableau trié ?

Exercice 5.11. Rappeler quelle est la complexité d'une fonction de recherche "naïve" de la forme :

```
int recherche(int n,int T[],int x) { ... }
```

Comment peut-on écrire une fonction plus efficace lorsqu'on suppose que le tableau passé en paramètre est trié ? Que pensez-vous de la fonction suivante :

```
int recherche_rapide(int n,int T[],int x) {
  trier(n,T);
  return recherche_efficace(n,T,x);
}
```

Exercice 5.12. On dispose de deux tableaux T et U de tailles respectives n et m et on voudrait connaitre les éléments de T qui n'apparaissent pas dans U d'une part et ceux de U qui n'apparaissent pas dans T d'autre part. Sous quelle forme proposez-vous de renvoyer le résultat ? Ecrire un programme de la forme :

```
void intrus(int n,int T[],int m,int U[], ??? ) { ... }
```

qui réalise ce travail. Quelle-est sa complexité ? On suppose maintenant que les tableaux sont triés, écrire alors un programme plus efficace :

```
void intrus_efficace(int n,int T[],int m,int U[], ??? ) { ... }
```

quelle est sa complexité ? Que pensez-vous du programme suivant :

```
void intrus_rapide(int n,int T[],int m,int U[], ??? ) {
  trier(n,T);
  trier(m,U);
  intrus_efficace(n,T,m,U, ??? );
}
```

Exercice 5.13. Ecrire et évaluer la complexité d'un programme éliminant les répétitions dans un tableau.

Exercice 5.14. L'arbre de la figure 1 représente un programme qui prend en entrée un tableau `t` de taille 3, qui réalise des comparaisons entre certaines cases de ce tableau et retourne les éléments du tableau dans un ordre différent (les valeurs retournées sont encadrées) : Montrer que le tableau retourné par ce programme est trié. Ecrire le programme C qui correspond à cet arbre sous la forme :

```
void tri(int t[],int u[]) { ... }
```


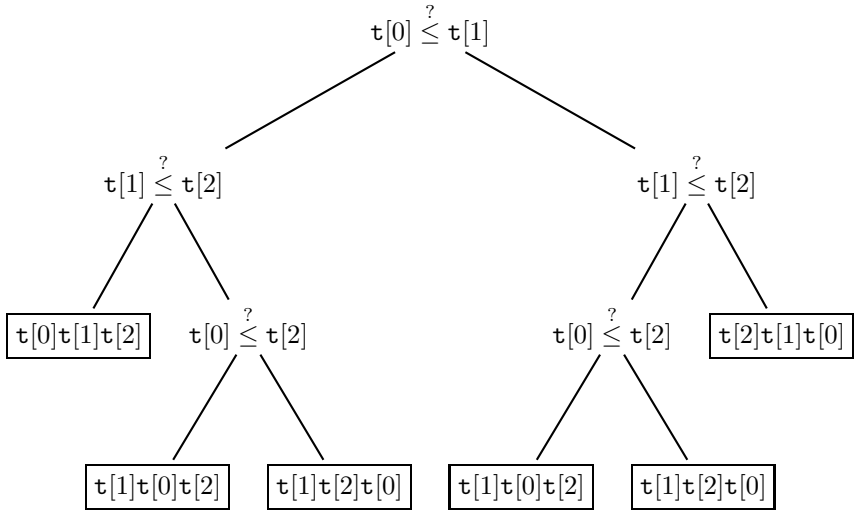

FIG. 1

On appelle *tri comparatif* un programme de tri dont le fonctionnement peut être représenté par un arbre de ce genre, une fois qu'on a fixé la taille des entrées. Les tris *génériques* (*ie* les tris qui ne font pas d'hypothèses sur les données qu'ils reçoivent) sont en général comparatifs.

Exercice 5.15. Montrer que le tri-bulles est un tri comparatif en détaillant son fonctionnement sur des tableaux de taille 3. Ecrire un programme de tri permettant de trier des tableaux ne contenant que des nombres entre 0 et 9 qui ne soit pas un tri comparatif.

Exercice 5.16. On considère ici un tri comparatif quelconque et on note $f(n)$ sa complexité *au pire* en fonction du nombre de *comparaisons*. Que représente $f(n)$ sur l'arbre qui décrit le tri ? Montrer que cet arbre doit posséder au moins $n!$ feuilles. Montrer alors que :

$$2^{f(n)} \geq n!$$

(indication : combien de feuilles peut posséder un arbre binaire dont la plus longue branche a pour hauteur h ?). Montrer enfin que :

$$n \log(n) = O(f(n))$$

On a ainsi démontré que la complexité au pire d'un tri comparatif est supérieure à $n \log(n)$ comparaisons. Y-a-t-il des programmes qui permettent d'atteindre effectivement cette complexité ?

Le résultat de l'exercice précédent peut donner de précieuses indications du la complexité d'autres programmes.

Exercice 5.17. Supposons que l'on se donne un ensemble fini de n points du plan $M_0, \ldots, M_{n-1}$. *L'enveloppe convexe* de ces points est le plus petit polygone convexe contenant tous ces points (si ces points sont des clous, l'enveloppe convexe est le polygone que l'on obtient en entourant ces clous d'un élastique). Quels seraient les paramètres d'une procédure de calcul d'enveloppe convexe ? Montrer comment on pourrait, à partir d'une telle procédure, construire un programme de tri. En supposant que ce programme de tri est comparatif, en déduire que la complexité *au pire* du calcul de l'enveloppe convexe est au moins $n\log(n)$.

Exercice 5.18. Cet exercice a pour but de montrer que, dans des cas particuliers (par exemple en faisant des hypothèses supplémentaires sur les données contenues dans le tableau) on peut trier plus rapidement que $n\log(n)$.

- Ecrire un programme de complexité linéaire permettant de trier les tableaux ne contenant que des entiers compris entre 0 et k.
- Comment effectuer *en une seule passe* le tri d'un tableau dont on sait qu'il ne peut contenir que des 1, des 2 et des 3 ?

6 Listes

Les quatres exercices suivants présentent différentes manières d'implémenter les listes. Pour chaque implémentation, on s'intéressera essentiellement aux fonctions suivantes :

- `consulter` et `affecter` qui permettent de consulter et modifier un élément stocké à un endroit précis de la liste,
- `inserer_apres` et `supprimer_apres` qui permettent d'ajouter un élément à un certain endroit de la liste, ou de le supprimer,
- `rechercher` et `afficher` qui permettent de tester si un entier est présent dans une liste ou d'afficher le contenu d'une liste.

De plus, on ne considèrera que des listes d'entiers.

Exercice 6.1. On représente ici une liste au moyen d'une suite de cellules. Chaque cellule contient l'entier stocké ainsi qu'un pointeur sur la cellule suivante. Une liste est représentée au moyen d'un pointeur sur la cellule de tête. Définir les types `Cellule` et `Liste`. Implémenter ensuite les fonctions :

```
int consulter(Liste* l, int i, int *x)
int affecter(Liste* l, int i, int x)
```


(chaque fonction retourne un entier, nul en cas d'erreur (l'indice i est hors des limites de la liste). Les indices fonctionnent de la même manière que pour les tableaux : si la liste contient n cellules, elles portent les indices $0, \ldots, (n-1)$). Les fonctions suivantes permettent de tirer parti du caractère dynamique des listes (comparées au tableaux) :

```
int inserer_apres(Liste* l, int i, int x)
int supprimer_apres(Liste* l,int i)
```

(on passera -1 en paramètre a `inserer_apres` et `supprimer_apres` afin d'obtenir des insersions et suppressions en tête de liste). Enfin deux fonctions nécessitant de parcourir une liste :

```
int rechercher(Liste* l, int x)
void afficher(Liste* l)
```

(la fonction `rechercher` retourne la première position où `x` apparait dans la liste, -1 si il n'y apparait pas). Calculer la complexité de ces fonctions.

Exercice 6.2. On choisit maintenant de ne plus utiliser des entiers pour repérer les positions des cellules dans la liste, mais tout simplement des pointeurs sur les cellules elles-mêmes. Implémenter les fonctions :

```
int consulter(Liste* l, Cellule* c, int *x)
int affecter(Liste* l, Cellule* c, int x)
int inserer_apres(Liste* l, Cellule* c, int x)
int supprimer_apres(Liste* l,Cellule* c)
```

(on passera la cellule nulle (pointeur `NULL`) en paramètre à `inserer_apres` et `supprimer_apres` pour représenter des insersions et suppressions en tête de liste). Calculer leur complexité. Comparer avec les résultats obtenus dans l'exercice précédent.

Exercice 6.3. Plutôt que d'utiliser un type différent pour les cellules et les listes, on définit le type `Liste` par :

```
typedef Cellule Liste
```

(ceci revient à représenter les listes au moyen d'une cellule, dite fictive, placée en tête de la séquence de cellules contenant les éléments de la liste). Ainsi, toute liste (même vide) contient au moins une cellule. Implémenter à nouveau les fonctions :

```
int consulter(Liste* l, Cellule* c, int *x)
int affecter(Liste* l, Cellule* c, int x)
int inserer_apres(Liste* l, Cellule* c, int x)
int supprimer_apres(Liste* l,Cellule* c)
```

et comparer cette implémentation avec celle de l'exercice précdent.

Exercice 6.4. Implémenter les fonctions :

```
int consulter(Liste* l, int i, int *x)
int affecter(Liste* l, int i, int x)
int inserer_apres(Liste* l, int i, int x)
int supprimer_apres(Liste* l,int i)
int rechercher(Liste* l, int x)
void afficher(Liste* l)
```

en utilisant la structure liste de l'exercice précédent (avec cellule fictive en tête), mais en privilégiant un traitement récursif.

Exercice 6.5. On cherche à reconnaître les listes palindromes, c'est-à-dire les listes identiques à leur miroir. Trouver une représentation chaînée judicieuse et écrire un programme reconnaissant les palindromes.

Exercice 6.6. Pour tout polynôme P, on note n le nombre de monômes de degrés distincts, et $e_1 < \ldots < e_n$ et $a_1, \ldots, a_n$ les coefficients tels que $P = a_1 x^{e_1} + \ldots + a_n x^{e_n}$. Trouver une représentation adaptée par liste chaînée et écrire des procédures pour additionner, multiplier et dériver les polynômes.

Exercice 6.7. On raconte que l'historien juif connu Flavius Josèphe sauva sa vie dans le premier siècle de notre ère de la façon que voici. La ville de Jotapate ayant été prise par les troupes romaines de l'empereur Vespasien, il se réfugia avec 40 de ses coreligionnaires dans une caverne. Ceux-ci, à l'exception de l'un d'entre eux, résolurent de se tuer pour ne pas tomber entre les mains de leurs vainqueurs. Josèphe, ne pouvant les en dissuader et ne voulant pas être entrainé dans un massacre général, imagina, parait-il, de mettre de l'ordre dans cette tuerie. Il fit placer tout le monde en cercle, en mettant habilement le seul de ses compagnons, qui, comme lui, n'était pas décidé à mourrir, au $x^{\text{ème}}$ rang et se mettant non moins habilement au $y^{\text{ème}}$. Ceci fait, on convint, pour lui faire plaisir, de compter les vivants de 3 en 3 et d'égorger chaque fois l'homme qui serait ainsi désigné. Le dernier devait se suicider pour que personne ne survive. Grâce à ces dispositions habiles, Josèphe et son compagnon restèrent les deux dernier et naturellement ne respectèrent pas les conventions

établies, grâce à quoi ils purent sauver leur vie. Ecrire un programme permettant de calculer le choix de x et y qui a permis a Josèphe de sauver sa vie, en remplaçant 40 et 3 par des entiers n et m quelconques.

7 Piles et files

Exercice 7.1. Implémenter une structure de pile en utilisant une représentation chaînée.

Exercice 7.2. Un train est composé de wagons qui ont été collectés dans plusieurs gares. Afin de redistibuer ces wagons dans d'autres gares, on voudrait réordonner la composition du train de sorte que les wagons soient placés dans l'ordre dans lequel ils vont être distribués. On dispose pour cela d'une voie munie d'un aiguillage vers deux autres voies en cul-de-sac. Expliquer comment procéder.

Exercice 7.3. `F` étant un pointeur sur une *file* d'entiers et `x`, `y`, `z` des pointeurs sur des entiers, dire ce qu'imprime le segment de code suivant :

```
init-file(F);
ajouter(F,5); ajouter(F,6); ajouter(F,7); ajouter(F,8);
retirer(F,x); retirer(F,y); ajouter(F,*x+1);
ajouter(F,*y+1);
retirer(F,x); ajouter(F,*y);
while !(file-vide(F))
    {
        retirer(F,x);
        printf("%d\n",x);
    }
```

Même question en supposant que `F` pointe sur une *pile*.

Exercice 7.4. Une file à double sens est une structure de données linéaire pour laquelle les insertions et les suppressions peuvent se faire aux deux extrémités. Donner une implémentation par pointeurs pour un tel objet.

8 Arbres

Exercice 8.1. Donner les représentations par tableau et par liste de fils de

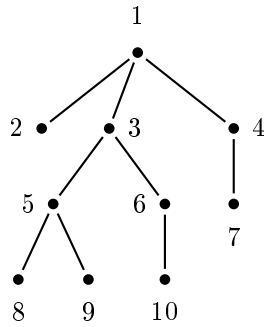

FIG. 2

l'arbre représenté figure 2

Exercice 8.2. Ecrire pour l'arbre de la figure 2 les représentations linéaires R_1 et R_2 définies de la façon suivante : soit A un arbre de racine r et de sous-arbres $A_1, \ldots, A_p$, alors

$R_1(A) = r\ [R_1(A_1)]\ \ldots\ [R_1(A_p)]$ et $R_2(A) = r\ [R_2(A_1)\ ,\ \ldots\ ,\ R_2(A_p)]$.

Ecrire une fonction qui, recevant un arbre représenté par listes de fils, en affiche une représentation linéaire.

Exercice 8.3. Appliquer la bijection "fils aîné – frère droit" à l'arbre de la figure 2 pour le représenter par un arbre binaire.

On cherche à effectuer un certain nombre d'opérations concernant un arbre général A en ne se servant que de sa représentation binaire B.

- Que représentent respectivement le bord gauche et le bord droit de l'arbre binaire B vis à vis de A ?
- Ecrire une fonction qui donne la liste des fils d'un nœud de A.
- Comment simuler un parcours préfixe de A ? Et un parcours suffixe ?
- Comment calculer la hauteur de A ?

Exercice 8.4. Ecrire des fonctions pour calculer la hauteur d'un arbre binaire et le nombre de nœuds qu'il contient.

Exercice 8.5. Ecrire une fonction qui teste l'égalité de deux arbres binaires.

On dit qu'un arbre binaire A figure dans un arbre binaire B s'il est égal à B ou à l'un de ses sous-arbres. Ecrire une fonction récursive qui teste si A figure dans B.

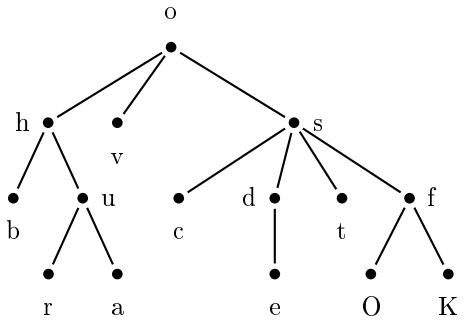

FIG. 3

Exercice 8.6. L'occurrence d'un nœud dans un arbre est un mot sur l'alphabet $\{0,1\}$ décrivant le chemin de la racine à ce nœud. L'occurrence de la racine est le mot vide ϵ, et si un nœud a pour occurrence w, ses fils gauche et droit ont respectivement pour occurrence $w0$ et $w1$. On peut ainsi décrire un arbre (non étiqueté) par l'ensemble des occurrences de ses nœuds.

- Quel est l'ensemble décrivant l'arbre ci-dessous ?
- Quel est l'arbre décrit par $\{\epsilon, 0, 1, 00, 10, 11, 000, 001, 101, 110, 0011\}$?
- Exprimer le bord gauche et le bord droit d'un arbre, le père et le fils d'un nœud, la hauteur d'un nœud et celle de l'arbre.

Exercice 8.7. Situer sur l'exemple de la figure 2 les différents traitements possibles lors d'un parcours général à main gauche :

- $F(f)$: le traitement à faire sur la feuille f
- $P(n)$: le traitement préfixé du nœud interne n (avant ses fils)
- $T_i(n)$: le traitement au i-ème passage *intermédiaire* au nœud n
- $S(n)$: le traitement suffixé de n (après tous ses fils).

Donner l'ordre dans lequel ces différents traitements seront effectués.

Exercice 8.8. Considérons l'arbre de la figure 3. En reprenant le parcours général récursif d'un arbre, indiquer ce qui s'affiche à l'écran dans les cas suivants :

	F	P	T_1	T_2	T_i $(i \geqslant 3)$	S
cas 1	rien	affichage	rien	rien	rien	rien
cas 2	affichage	rien	affichage	rien	rien	rien
cas 3	affichage	rien	rien	affichage	rien	rien
cas 4	affichage	rien	rien	rien	rien	affichage

Exercice 8.9.
- Quels sont les arbres binaires sur lesquels les ordres préfixe et infixe coïncident ? Et les ordres préfixe et suffixe ?
- Quels arbres binaires arithmétiques ont $1+2*3-4$ pour liste de nœuds en ordre infixe ? Donner pour chacun les expressions infixe bien parenthésée, préfixe et suffixe correspondantes.
- Montrer que, si l'on connait la liste des nœuds d'un arbre binaire en ordre préfixe ET en ordre infixe, on peut reconstruire l'arbre.

Exercice 8.10. Expliquer comment on peut représenter une expression arithmétique de la forme :

$$(3+7)-4*(10+6)$$

par des arbres binaires. Comment définir et manipuler de tels arbres en C ? Etant donnée une expression représentée par un arbre binaire, on veut l'évaluer en utilisant un pile. Proposer une fonction :

```
void evaluer(Expr e,Pile *P)
```

permettant de réaliser ce travail.

Exercice 8.11. On peut utiliser des arbres pour représenter de manière compacte des ensemble de mots. L'arbre de la figure 4 par exemple représente l'ensemble de mots $\{maison, mais, mars, mille, port\}$. Pourquoi certains nœuds sont-ils différenciés ? Proposer une structure d'arbre adaptée et écrire une fonction :

```
Arbre* compacte(int n, char* T[])
```

qui transforme un tableau de chaînes de caractères en arbre.

9 ABR et AVL

Exercice 9.1. Ecrire des fonctions :
- pour afficher les éléments d'un arbre binaire de recherche dans l'ordre,
- pour chercher un élément dans un arbre binaire de recherche.

Exercice 9.2. Soit A et B deux arbres binaires de recherche. On dit que A est *de domaine plus grand* que B si tous les éléments de B sont compris entre le plus petit et le plus grand élément de A, et que A *contient* B si tous les éléments de B sont dans A.

Ecrire des fonctions qui testent ces deux propriétés.

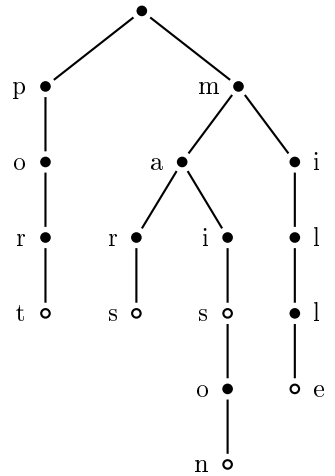

FIG. 4

Exercice 9.3. Ecrire la fonction d'ajout d'un élément comme feuille d'un arbre binaire de recherche, et déterminer les arbres créés à partir de l'arbre vide par les ajouts successifs de :
- 5, 7, 2, 4, 3, 6, 1,
- 1, 2, 3, 4, 5, 6, 7,
- 4, 2, 1, 6, 7, 5, 3.

Pour chacun de ces exemples, donner, s'il en existe, d'autres ordres d'insertion aboutissant au même arbre.

Plus généralement, on peut montrer que le nombre de façons d'obtenir un arbre A de taille n par ajouts successifs de feuilles est égal au quotient de $n!$ par le produit des tailles des sous-arbres de A.

Exercice 9.4. Ecrire une fonction `int suppmax(abr * A)` qui rend la valeur du plus grand élément de A et le supprime.

Ecrire une fonction `void supprime(int x, abr * A)` qui supprime l'élément x de A en utilisant la procédure précédente.

Faire tourner `supprime(5,A)` sur l'arbre de la figure 5.

Exercice 9.5. Faire tourner `supprime(k,A)` puis `supprime(m,A)` sur l'arbre A de la figure 6. Montrer que l'arbre obtenu après une suite de suppressions dans un arbre donné ne dépend pas de l'ordre de ces suppressions.

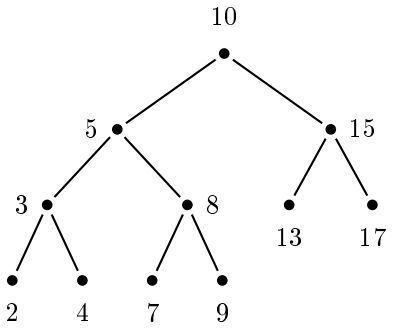

FIG. 5

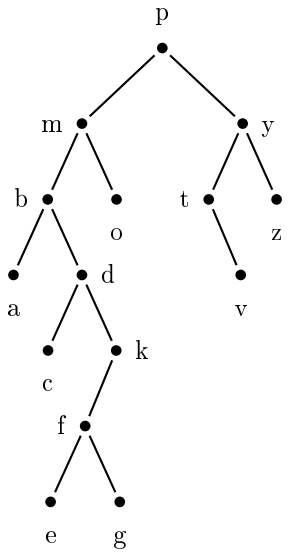

FIG. 6

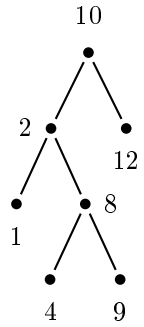

FIG. 7

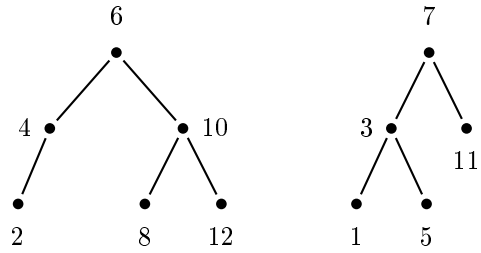

FIG. 8

Exercice 9.6. Ecrire une fonction `couper` qui prend en entrée un arbre binaire de recherche `A` et un élément `elt` et fabrique deux arbres binaires de recherche `G` et `D` contenant respectivement les éléments de `A` inférieurs et supérieurs à `elt`.

Faire tourner `couper(A,6,G,D)` sur l'arbre A de la figure 7. Ecrire une fonction permettant d'ajouter un élément à la racine d'un arbre binaire de recherche, et déterminer les arbres créés à partir de l'arbre vide par les ajouts successifs donnés par les listes 5 7 2 4 3 6 1, 1 2 3 4 5 6 7, et 4 2 1 6 7 5 3.

Exercice 9.7. Ecrire une fonction `fusion(abr * A, abr * B, abr * C)` en utilisant la fonction `couper`, et l'utiliser pour fusionner les arbres de la figure 8.

Exercice 9.8. Indiquer la valeur du déséquilibre de chaque nœud des arbres de la figure 9.

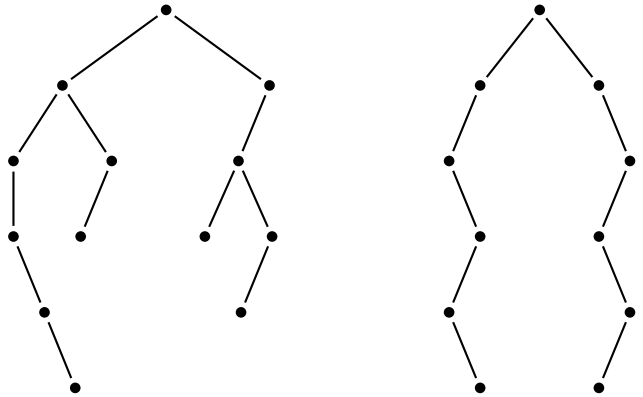

FIG. 9

Exercice 9.9. Caractériser les arbres h-équilibrés qui ont le plus de nœuds pour une hauteur donnée h. Combien ont-il de nœuds ? Que peut-on dire du paramètre `deseq` sur les nœuds de tels arbres ?

Mêmes questions pour les arbres qui ont le moins de nœuds pour une hauteur donnée. Que se passe-t-il si on enlève une feuille à un tel arbre ?

Exercice 9.10. Décrire les opérations de rotation simple (`rg`, `rd`) et double (`rgd`, `rdg`), et montrer qu'elles peuvent être programmées en temps constant (*i.e.* indépendant de la taille de l'arbre concerné).

Expliquer comment elles agissent sur le déséquilibre d'un arbre, en déterminant en particulier quels sont les noeuds dont le déséquilibre est modifié.

Exercice 9.11. Ecrire l'opération de rééquilibrage d'un nœud `reequil` en utilisant le paramètre `deseq`.

Exercice 9.12. On veut écrire une fonction `ajout-avl` qui ajoute un élément dans un arbre AVL avec les spécifications suivantes :

```
struct AVL {
             int deseq;
             int val;
             struct AVL * g, * d;
           };
```

`ajout-avl` : Arbre $\times$ Elément $\longmapsto$ Arbre.

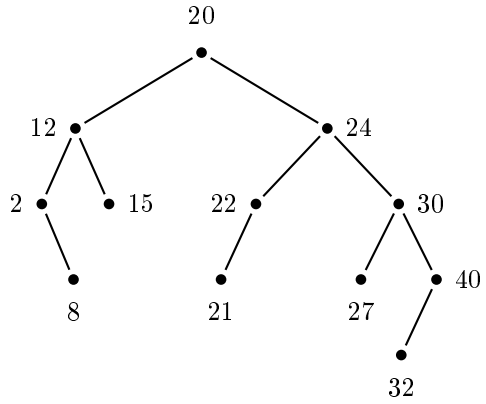

FIG. 10

On rappelle que le principe de l'algorithme itératif d'ajout dans un AVL est de :

- descendre dans l'arbre pour trouver la position où doit se faire l'ajout, en mémorisant le dernier nœud v de ce chemin pour lequel le déséquilibre vaut ± 1,
- effectuer l'ajout (comme feuille),
- modifier les valeurs du champ `deseq` sur le chemin de v à la feuille,
- rééquilibrer éventuellement l'arbre par une rotation en v.

Exercice 9.13. Créer un arbre AVL par adjonction successive des éléments

7, 1, 4, 10, 2, 5, 9, 3, 12, 8, 6, 11.

Exercice 9.14. En utilisant l'opération de rééquilibrage définie dans le premier exercice de la secyion et les opérations supposées définies :

`max`	:	Arbre $\longmapsto$ Elément	qui rend le plus grand élément de l'arbre,
`suppmax`	:	Arbre $\longmapsto$ Arbre	qui le supprime,

Décrire l'opération de suppression d'un élément quelconque `supp-avl`.

Exercice 9.15. Supprimer la valeur 20 dans l'arbre de la figure 10.

10 Tables de hachage

Le but est d'étudier différentes manières de représenter des ensembles d'objets (appartenant tous à un ensemble U) au moyen de tables de hachage. Pour fixer les idées, on suppose que :

- U est l'ensemble des chaînes de caracteres,
- on a une fonction de hachage :
  `int h(char* s)`
  telle que $h(s)$ est toujours compris entre 0 et $M-1$ avec $M = 37$ (par exemple).

On sait qu'une telle fonction de hachage ne peut éviter de créer des *collisions* et on propose deux méthodes pour les résoudre. Dans chaque cas, on demande d'implémenter les fonctions permettant de créer une table vide, d'insérer et supprimer des éléments dans cette table et rechercher si un élément y est présent.

Exercice 10.1. On utilise ici un tableau de taille M et chaque case $T[i]$ du tableau pointe sur une liste contenant les éléments dont le code de hachage est i qui ont été jusqu'à présent stockés dans cette table. Implémenter les différentes fonctions de manipulation d'une telle table.

Exercice 10.2. Calculer la complexité en moyenne d'une recherche dans une telle table (en faisant si nécessaire des hypothèses sur la fonction de hachage).

Exercice 10.3. Quelles sont les qualités que doit posséder une "bonne" fonction de hachage ? Donner un ou deux exemples dans ce cas précis.

Exercice 10.4. On veut maintenant éviter de manipuler des listes et utiliser une case du tableau pour chaque donnée à stocker. Pour résoudre les collisions, on propose d'utiliser m fonctions de hachage $h_0, \ldots, h_{m-1}$ et l'algorithme suivant pour l'insertion :

- si $T[h_0(x)]$ est vide, alors placer x dans $T[h_0(x)]$,
- sinon, si $T[h_1(x)]$ est vide, alors placer x dans $T[h_1(x)]$,
- ...

Expliquer pourquoi, à x fixé, $(h_0(x), \ldots, h_{m-1}(x))$ doit être une permutation de $(0, \ldots, m-1)$. Quelles sont les limitations de cette approche ? Implémenter les différentes fonctions de manipulation d'une telle table.

Exercice 10.5. Quelle est la complexité moyenne de la recherche dans une telle table ?

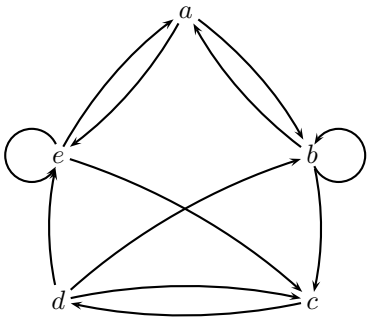

FIG. 11

Exercice 10.6. Proposer une méthode permettant de calculer les fonctions de hachage $h_0, \ldots, h_{m-1}$.

11 Algorithmes de graphes

Exercice 11.1. On considère le graphe de la figure 11.

- Donner les représentations de ce graphe par listes d'adjacence et par matrice d'adjacence, indiquer le degré entrant et sortant de chaque sommet.
- Donner des exemples de chemins et de circuits.
- Enumérer les circuits élémentaires.

Exercice 11.2. On considère le graphe de la figure 12. Calculer les composantes fortement connexes de ce graphe ainsi que le graphe réduit. Que se passe-t-il si on remplace $e \to g$ par $g \to e$?

Exercice 11.3. On considère le graphe de la figure 13. Quels sont les sommets entre lesquels il existe un chemin de longueur 2 ? Calculer la matrice d'adjacence A de ce graphe puis la matrice A^2. Montrer que pour $n \geq 1$, le coefficient (i, j) de A^n représente le nombre de chemins de longueur n entre i et j.

Exercice 11.4. Déterminer le nombre de graphes orientés (respectivement non orientés) à n sommets, d'abord dans le cas où les boucles sur un même sommet sont autorisées puis dans le cas où elles ne le sont pas. Calculer le

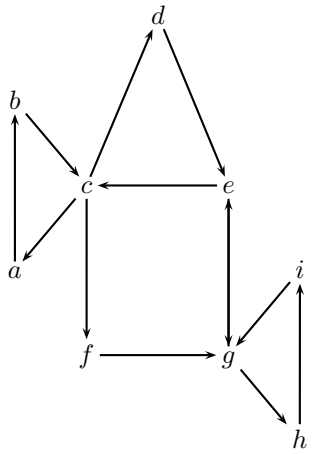

FIG. 12

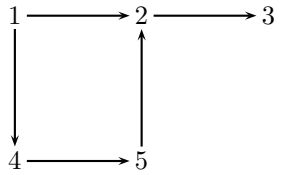

FIG. 13

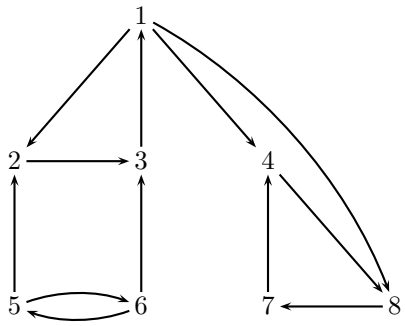

FIG. 14

nombre d'arêtes du graphe orienté complet à n sommets.

Exercice 11.5. Soit G un graphe non orienté de sommets $s_1, \ldots, s_n$. Montrer que $\sum_{i=1}^{n} d(s_i)$ est paire. En déduire que G a un nombre pair de sommets de degré impair. **Exercice 11.6.** Faire tourner l'algorithme de parcours en profondeur sur le graphe de la figure 14. Donner le sous-graphe de liaison associé et déterminer les arcs de liaison, les arcs retour, les arcs avant et les arcs couvrants. Calculer les composantes fortement connexes de G.

Exercice 11.7. On suppose donné un graphe G représenté par sa matrice d'adjacence. Ecrire un programme permettant de réaliser le parcours en profondeur de G.

Exercice 11.8. Le matin, il faut :
- Prendre une douche
- Préparer un café
- Se réveiller
- Choisir ses vêtements
- Se lever
- Boire un café
- S'habiller
- Sortir
- Préparer ses affaires

Représenter ces différentes activités sous forme d'un graphe. Proposer une méthode permettant de décider dans quel ordre on doit les effectuer.

Exercice 11.9. Faire tourner l'algorithme de parcours en largeur sur le graphe de l'exercice 1.

Exercice 11.10. On suppose donné un graphe G représenté par sa matrice d'adjacence. Ecrire un programme permettant de réaliser le parcours en largeur de G.

Exercice 11.11. On considère le jeu suivant : à chaque tour, on dispose d'un entier i sur lequel on peut effectuer l'une des deux opérations suivantes :

- Faire une multiplication par 2
- Faire une division entière par 3 (à condition que i soit lui-même plus grand que 3)

On se donne également un entier k. Le jeu consiste à essayer d'atteindre k à partir de 1 en utilisant seulement ces deux opérations. Proposer une méthode qui permet de trouver la suite d'opérations faisant passer de 1 à k (du moins, lorsqu'une telle suite existe). Que donne cette méthode dans le cas où une telle suite n'existe pas ?

Soit G un graphe non orienté connexe. Le but de ce TD est de montrer le résultat suivant :

- G possède un chemin eulérien si, et seulement si, le nombre de sommets de G de degré impair est 0 ou 2,
- dans le cas où G ne possède aucun sommet de degré impair, G possède même un cycle eulérien.

Exercice 11.12. Supposons tout d'abord que G possède un cycle eulérien, montrer qu'alors G n'a pas de sommet de degré impair. Si maintenant G possède un chemin eulérien, mais pas de cycle eulérien, montrer qu'alors G possède deux sommets de degré impair.

Exercice 11.13. On suppose que G ne possède aucun sommet de degré impair. On veut montrer par l'absurde que G possède un chemin eulérien. On suppose donc que G ne possède pas de chemin eulérien. Ainsi, aucun chemin simple ne contient toutes les arêtes de G. On note M le nombre maximum d'arêtes que peut contenir un chemin simple de G et on considère un chemin simple :

$$w = s_0, \ldots, s_M$$

contenant M arêtes.

- Montrer que $s_0 = s_M$.
- Montrer qu'il existe une arête de G située hors de w mais dont l'une des extrémités est située sur le parcours de w.

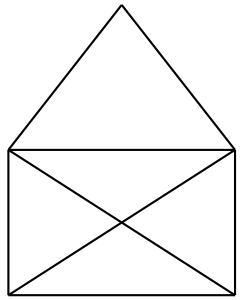

FIG. 15

- Montrer qu'il existe un chemin simple de G contenant $M+1$ arêtes et conclure.

On en déduit donc que G possède un chemin eulérien, notons s_0 le sommet de départ et s_m le sommet d'arrivée. Montrer que $s_0 = s_m$ et en déduire que G possède un cycle eulérien.

Exercice 11.14. On suppose que G contient deux sommets de degré impair. En utilisant l'exercice précédent, montrer que G possède un chemin eulérien.

Exercice 11.15. Beaucoup de problèmes, *a priori* différents, peuvent se ramener à la recherche ou a l'existence de parcours eulériens dans un graphe. A titre d'exemple :

- peut-on dessiner la figure 15 d'un seul trait et sans repasser deux fois sur le même trait ?
- peut-on disposer tous les dominos d'un jeu en respectant les règles et de manière à ne former qu'une seule ligne ?

Exercice 11.16. Un graphe possédant un chemin eulérien est-il toujours connexe ?

On considère dans la suite un graphe orienté $G = (S, A)$ muni d'une fonction de coût (c'est à dire un fonction à valeurs positives définie sur A). Pour tout chemin w dans G, on appelle coût de w et on note $c(w)$ la somme des coûts des arcs de w. Soient u et v des sommets de G. Si il existe un chemin entre u et v, on note $\delta(u, v)$ le coût le plus petit d'un chemin entre u et v. Dans le cas contraire, on dit que $\delta(u, v)$ est infini.

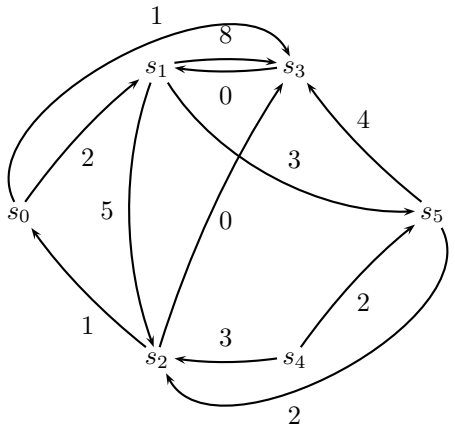

FIG. 16

Exercice 11.17. Soit :

$$w = s_0 \to s_1 \to \ldots \to s_n$$

un plus court chemin entre deux sommets s_0 et s_n. Montrer que quels que soient i et j avec $0 \leq i < j \leq n$, le chemin de s_i à s_j obtenu à partir de w est un plus court chemin. Soit s un sommet de G, montrer que pour toute arête (u, v) appartenant à A on a :

$$\delta(s, v) \leq \delta(s, u) + c(u, v)$$

Exercice 11.18. Faire tourner l'algorithme de Dijkstra sur le graphe de la figure 16.

Exercice 11.19. Soit s_0 un sommet de G et supposons que l'on ait partitionné l'ensemble S des sommets de G en deux sous-ensembles S' et S'' tels que :

- S' contient s_0 et pour tout sommet u de S', $\delta(s_0, u)$ est connu,
- pour tout v de S'' on connait la longueur d'un plus court chemin de s_0 à v qui ne sort pas de S' (c'est à dire dont tous les sommets sauf le dernier sont dans S') et on note cette longueur $d(v)$.

Montrer que si v est un sommet de S'' tel que $d(v)$ est le plus petit possible, alors $d(v) = \delta(s_0, v)$. Montrer que l'on peut alors retrouver ainsi l'algorithme de Dijkstra. Calculer sa complexité.

Exercice 11.20. On suppose ici que G ne contient pas de cycle. Soit s_0 un sommet de G sans prédécesseur et on suppose construit un sous ensemble S' de S contenant s_0 et tel que pour chaque u de S' :

- on connait $\delta(s_0, s)$,
- tous les prédécesseurs de u sont dans S'.

On suppose que S' est strictement inclus dans S. Montrer qu'il existe alors un sommet v situé hors de S' tel que tous les prédécesseurs de v sont dans S'. Que vaut alors $\delta(s_0, v)$? En déduire un algorithme de calcul des plus courts chemins d'origine s_0. Que se passe-t-il lorsque l'on autorise des coûts négatifs ?

Le tableau suivant liste les différentes tâches qui interviennent dans la construction d'une maison. Pour chaque tâche, on indique sa durée ainsi que les tâches qui doivent la précéder :

Code	Libellé	Durée (semaines)	Prédécesseurs
1	Maçonnerie	7	aucun
2	Charpente de la toiture	3	1
3	Toiture	1	2
4	Plomberie et electricité	8	1
5	Façade	2	3, 4
6	Fenêtres	1	3, 4
7	Aménagement du jardin	1	3, 4
8	Plafonds	3	6
9	Peintures	2	8
10	Emménagement	1	5, 7, 9

On a coutume d'ajouter deux tâches fictives de début et de fin, α et ω, de durée nulle. Quelle est la durée (au plus tôt) de construction de la maison ?

Exercice 11.21. On suppose ici que l'ensemble des sommets de G est l'ensemble $S = \{1, \ldots, n\}$. Pour k tel que $0 \leq k \leq n$, et $i, j \in S$, on note $A_k[i, j]$ la distance (éventuellement infinie) d'un plus court chemin de i à j ne passant par aucun sommet strictement supérieur à k (autre que i et j).

- Calculer $A_0[i, j]$.
- Pour $k \geq 1$, calculer $A_k[i, j]$ en fonction de A_{k-1}.
- En déduire un algorithme pour calculer les plus courts chemins dans G et le faire tourner sur le graphe de la figure 17. Quelle est sa complexité ?

Exercice 11.22. Considérons un ensemble fini N dont les éléments représentent les différents sites d'un réseau. Certains de ces sites sont reliés par une connexion c et cette connexion est fiable avec une probabilité f_c. Etant donnés deux points du réseau, comment trouver le chemin le plus fiable entre ces deux points ? (Indication : calculer la fiabilité d'un chemin (c'est à dire

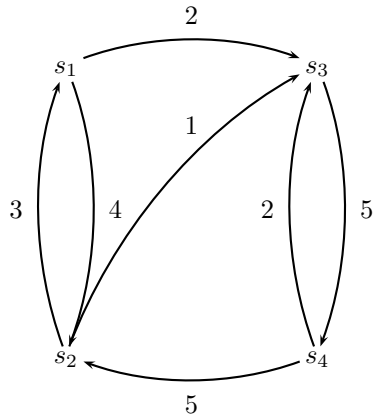

FIG. 17

sa probabilité de fonctionnement) en fonction de la fiabilité des arêtes qui le composent).

Exercice 11.23. Sur le bord d'une rivière se trouvent un batelier, une chèvre, un chou et un loup. Tous veulent traverser la rivière mais le bateau est si petit qu'il ne peut contenir qu'un seul objet en plus du batelier. Sachant qu'il ne faut jamais laisser ensemble le loup et la chèvre, ni la chèvre et le chou, combien faut-il que le batelier fasse d'aller-retours pour transporter tout le monde de l'autre coté ?

Exercice 11.24. Montrer qu'un arbre à $n \geq 2$ sommets possède au moins 2 sommets de degré 1.

Exercice 11.25. Montrer que si l'intersection de deux sous-arbres d'un arbre est non vide, alors cette intersection est elle-même un sous-arbre.

Exercice 11.26. On dit qu'un graphe non orienté $G = (S, A)$ est 2-colorable si on peut associer une couleur à chaque sommet de sorte que :
- au plus deux couleurs sont utilisées,
- deux sommets adjacents ne sont jamais de la même couleur.

On dit que G est biparti si on peut trouver deux sous-ensembles de S, S' et S'', tels que :
- S' et S'' sont disjoints et leur réunion est S,
- toute arête de G possède une extrémité dans S' et l'autre dans S''.

Montrer qu'être biparti est équivalent à être 2-colorable. Montrer qu'un arbre est toujours 2-colorable.

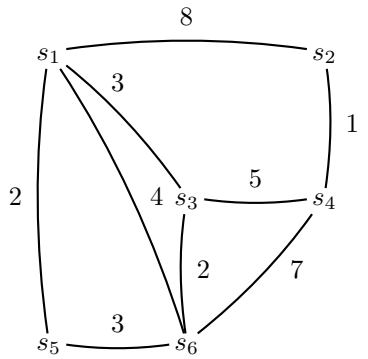

FIG. 18

Dans cette partie, $G = (S, A)$ désigne un graphe non orienté connexe muni d'une fonction de coût à valeurs positives.

Exercice 11.27. On suppose ici que la fonction de coût est constante, égale à 1. Quel est alors le poids d'un arbre couvrant minimal de G ?

Exercice 11.28. Soit U un ensemble non vide et strictement inclus dans S. On note E l'ensemble des arêtes de G dont une extrémité est dans U et l'autre hors de U et on considère parmi ces arêtes une dont le coût est minimum, que l'on note e. Montrer alors que parmi les arbres couvrants de poids minimal de G il en existe un contenant cette arête.

Soit maintenant G' un sous-graphe de G. On suppose que :
- G' est un arbre,
- G' est inclus dans un arbre couvrant minimal de G, T,
- U est l'ensemble des sommets apparaissant dans G'.

Montrer alors que parmi les arbres couvrants de poids minimal de G, il en existe un contenant G' et e. En déduire un algorithme de calcul d'un arbre couvrant de poids minimal et l'appliquer au graphe de la figure 18.

Exercice 11.29. Appliquer l'algorithme de Kruskal au graphe de la figure 18.

12 Annexes