# Algorithmique

http://alexandre-mesle.com

30 octobre 2008

# Table des matières

# Chapitre 1

# Complexité des algorithmes

Un algorithme, programmé dans un langage différent, ou lancé sur une machine différente, ne mettra pas le temps à exécuter. Il serait pourtant important de trouver un moyen de mesurer son temps d'exécution, pour le comparer par exemple à d'autres algorithmes. Nous nous intéresserons à un critère, appelé **complexité dans le pire des cas**, qui nous permettra d'évaluer les performances d'un algorithme en terme de temps d'exécution.

## 1.1 Borne asymptotique supérieure

Une suite $u$ est de **borne asymptotique supérieure** $v$ si pour $n$ assez grand, $u$ est majorée par $v$ à une constante près. Plus formellement,

**Définition 1.1.1** $u_n \in \mathcal{O}(v_n)$ *si* $\exists N \in \mathbb{N}, \exists k \in \mathbb{R}, \forall n \geq N, u_n \leq k.v_n$

Pour $n$ supérieur à $N$, $u$ est majorée par $k.v$, donc par $v$ à la constante $k$ près. Si $N$ et $k$ existent, alors $u_n \in \mathcal{O}(v_n)$. En pratique, on utilise la propriété suivante :

**Propriété 1.1.1** $u_n \in \mathcal{O}(v_n)$ *ssi* $\lim\limits_{n \longrightarrow +\infty} \dfrac{u_n}{v_n} = l$

Plus simplement, $u_n \in \mathcal{O}(v_n)$ si $\frac{u_n}{v_n}$ converge vers une constante (notée $l$ dans la propriété).

## 1.2 Mesure de temps d'exécution

Un algorithme prend en entrée une quantité de données que nous noterons en règle générale $n$, par exemple la taille d'un tableau, le nombre d'élément d'une liste chaînée, le nombre de noeuds d'un graphe, etc. Nous considérerons le nombre d'opérations $u_n$ exécuté par l'algorithme et tenterons de classifier $u$ en utilisant une fonction $v$. Ce critère est intéressant pour plusieurs raisons :

- Quelle que soit le temps que mette une machine particulière à exécuter une opération, nous compterons le nombre d'opérations.
- Se limiter au dénombrement des opérations n'est pas suffisant, ce qui nous intéresse vraiment est la façon dont le nombre d'opérations effectuées par l'algorithme croît en fonction du nombre $n$ de données.

Nous considérerons en règle générale le temps d'exécution dans le pire des cas. Quelques bornes asymptotiques supérieures sont à considérer avec un attention particulière :

### 1.2.1 Algorithmes en temps constant, $\mathcal{O}(1)$

Un algorithme est en temps constant si son temps d'exécution est indépendant du nombre de données. Par exemple, retourner le premier élément d'une liste chaînée. En effet, que la liste contienne un seul élément ou $10^{10}$ éléments, le temps d'exécution de l'algorithme ne variera pas en fonction de $n$.

### 1.2.2 Algorithmes de complexité linéaire, $\mathcal{O}(n)$

Un algorithme est de complexité linéaire si le temps de calcul croît de façon linéaire en fonction du nombre de données. Si vous traitez deux fois plus de données, le temps d'exécution sera multiplié par deux. Par exemple, la recherche séquentielle dans un tableau non trié est de complexité linéaire. En effet, on effectue une recherche en parcourant le tableau avec une boucle (ou un algorithme récursif). Chacune des itérations se fait en temps constant, il suffit donc de compter le nombre d'itérations. Dans le pire des cas, l'élément que l'on recherche est celui qui est examiné en dernier. Il est donc nécessaire d'effectuer $n$ itérations.

### 1.2.3 Algorithmes de complexité logarithmique, $\mathcal{O}(logn)$

Un algorithme de complexité algorithmique croit de façon linéaire en fonction de $2^n$, cela signifie que pour doubler le temps d'exécution, il faut mettre au carré le nombre de données. Par exemple, la recherche par dichotomie dans un tableau trié. Si le tableau contient $2^{30}$ éléments, il faudra à peu près 30 itérations pour trouver l'élément recherché, contre $2^{30}$ avec une recherche séquentielle. Pour la plupart, ces algorithmes sont dans la pratique très performants.

### 1.2.4 Algorithmes de complexité quadratique $\mathcal{O}(n^2)$

Certains algorithmes de tri, par exemple, sont construits avec deux boucles imbriquées, et effectuent un nombre d'itérations de l'ordre de $n^2$. Bon nombre de problèmes pour lesquels il existe des algorithmes quadratiques admettent aussi des algorithmes de meilleure complexité.

### 1.2.5 Algorithmes de complexité $\mathcal{O}(n.logn)$

D'autres algorithmes de tri sont de complexité $\mathcal{O}(n.logn)$, c'est le meilleur résultat qu'il est possible d'obtenir avec des tris de comparaisons.

### 1.2.6 Algorithmes de complexité exponentielle $\mathcal{O}(k^n), k > 1$

Ces algorithmes sont tellement longs à l'exécution, qu'on ne les utilise presque jamais. Malheureusement, il existe des problèmes pour lesquels les seuls algorithmes de résolution exacte connus à l'heure actuelle sont de complexité exponentielle.

### 1.2.7 Algorithmes de complexité polynomiale $\mathcal{O}(n^k)$, $k$ fixé

Nous appelons ainsi tous les algorithmes dont le temps d'exécution peut être majoré par un polynôme. Nous utiliserons le terme polynomial par opposition à exponentiel. En algorithmique, il sera très important de faire la différence entre les algorithmes polynomiaux, utilisables en pratique, et les algorithmes exponentiels, inutilisables en pratique.

#### Exercice 1 - Tableaux

Déterminer la complexité dans le pire cas (précisez quel est ce pire cas) des opérations suivantes sur un tableau $t = [t_1, \ldots, t_n]$ à $n$ éléments non triés,

1. Affichage du $k$-ème élément de $t$.
2. Affichage des éléments de $t$.
3. Calcul de la somme cumulée des éléments de $t$.
4. Décalage vers la droite de la tranche $[t_k, \ldots, t_n]$ et insertion d'un élément au rang $k$.
5. Décalage vers la gauche de la tranche $[t_{k+1}, \ldots, t_n]$ pour supprimer $t_k$.
6. Recherche séquentielle d'un élément.
7. Vérification de l'appartenance de chaque élément de $t$ à un tableau $q$ non trié.

**Exercice 2 - Listes chaînées**

Déterminer la complexité dans le pire des cas (précisez de quel cas il s'agit) des opérations ci-dessous sur une liste chaînée $l = l_1 \longrightarrow l_2 \longrightarrow \ldots \longrightarrow l_n$ à $n$ éléments non triés. La seule information à notre disposition au début de l'exécution de chacun de ces algorithmes est un pointeur vers le premier élément de la liste. :

1. Affichage du $k$-ème élément de $l$.
2. Affichage des éléments de $l$.
3. Calcul de la somme cumulée des éléments de $l$.
4. Suppression du premier élément.
5. Suppression de l'élément de rang $k$, adresse du $(k-1)$-ème élément connue.
6. Suppression de l'élément de rang $k$, adresse du $(k-1)$-ème élément inconnue.
7. Recherche séquentielle d'un élément.
8. Vérification de l'appartenance de chaque élément de $l$ à une liste $q$ non triée.

**Exercice 3 - Tri par sélection**

Le tri par sélection d'un tableau $[t_1, \ldots, t_n]$ se fait comme suit. On fait varier $i$ de 1 à $n-1$. Pour chaque itération de $i$, on recherche dans la tranche $[t_i, \ldots, t_n]$ le plus petit élément et on l'échange avec $t_i$. Déterminer la complexité du tri par sélection.

**Exercice 4 - Hauteur minimale d'un arbre binaire**

Un arbre binaire est un arbre dont chaque noeud a 2 fils. Un arbre vide, donc sans noeud, noté $\emptyset$, est de profondeur $h(\emptyset) = -1$. Un noeud est une feuille si ses deux fils sont des arbres vides. Soit $x$ un noeud, $l$ et $r$ ses fils la hauteur de $x$ est alors $h(x) = 1 + \max(h(l), h(r))$. La hauteur d'un arbre binaire est la hauteur de son noeud de hauteur maximale, donc de la racine. La racine $r$ est de profondeur $p(r) = 0$, tout noeud $x$ de père $y$ est de profondeur $p(x) = p(y) + 1$.

1. Quelle est la hauteur d'une feuille d'un arbre binaire ?
2. Comment disposer les noeuds pour que la hauteur d'un arbre binaire à $n$ noeuds soit minimale ?
3. Nous ne considérerons dorénavant que des arbres binaires dans lesquels les noeuds sont disposés comme décrit dans la question précédente. Soit $A$ un arbre de profondeur $h$, déterminer pour tout $k \in \{0, \ldots, h-1\}$ le nombre de noeuds de $A$ dont la profondeur est $k$.
4. Encadrer le nombre de noeuds de profondeur $h$.
5. Donner la borne inférieure du nombre $n$ de noeuds d'un arbre binaire de profondeur $h$.
6. Majorer $h$ en fonction de $n$.
7. Donner une borne asymptotique supérieure de la hauteur $h$ d'un arbre binaire à $n$ noeuds de hauteur minimale.

# Chapitre 2

# Algorithmes itératifs

Un algorithme itératif est contruit avec des boucles, par opposition à récursif qui remplace les boucles par ds appels à lui-même.

## 2.1 Invariants de boucle

### 2.1.1 Le principe

On prouve la validité d'un algorithme itératif à l'aide de la méthode des **invariants de boucle**.

**Définition 2.1.1** *Un invariant de boucle est une propriété qui est vraie à chaque passage dans la boucle.*

On raisonne par récurrence pour montrer qu'une telle propriété est préservée à chaque itération de la boucle. Elle suffit à prouver la validité de l'algorithme si elle est trivialement vraie à la première itération et qu'à la dernière itération, sa véracité entraîne celle de l'algorithme. Voyons cela sur un exemple.

### 2.1.2 Exemple

On divise deux entiers strictement positifs $a$ et $b$ en déterminant deux entiers $q$ et $r$ tels que $a = bq + r$ avec $0 \leq r < b$. Voici un algorithme déterminant $q$ et $r$ :

```
q ⟵ 0
r ⟵ a
tant que r ≥ b
  | q ⟵ q + 1
  | r ⟵ r − b
fin tant que
```

On choisit comme invariant de boucle la propriété $a = bq + r$.

- Comme $q$ est initialisé à 0 et $r$ à $a$, alors la propriété $a = bq + r = b.0 + a$ est vérifiée avant le premier passage dans la boucle.
- Avant une itération arbitraire, supposons que l'on ait $a = bq + r$, montrons que cette propriété est conservée par cette itération. Soient $q'$ la valeur de $q$ à la fin de l'itération et $r'$ la valeur de $r$ à la fin de l'itération. Nous devons montrer que $a = bq' + r'$. On a $q' = q + 1$ et $r' = r - b$, alors $bq' + r' = b(q + 1) + (r - b) = bq + r = a$. La propriété est bien conservée.

## 2.2 Terminaison

### 2.2.1 Le principe

La démonstration ci-dessus est incomplète, d'une part on n'a pas démontré que le programme s'arrêtait, et de plus rien ne montre (pour le moment) que s'il s'arrête, on aura bien $0 \leq r < b$. On montre qu'un programme s'arrête en

prouvant que la séquence formée par les valeurs des variables au fil des itérations converge vers une valeur satisfaisant la condition d'arrêt.

### 2.2.2 Fin de l'exemple

- Commençons par montrer que le programme s'arrête. La séquence formée par les valeurs de $r$ au fil des itérations est strictement décroissante. Il y a donc nécessairement un moment où la valeur de $r$ sera strictement inférieure à celle de $b$.
- Maintenant, nous pouvons terminer la preuve de validité :
  - Si le programme s'arrête, c'est que la condition du `Tant que` n'est plus satisfaite, donc que $r < b$.
  - Il reste à montrer que $r \geq 0$. Comme $r$ est diminué de $b$ à chaque itération, si $r < 0$, alors à l'itération précédente la valeur de $r$ était $r' = r + b$, or $r' = r + b < b$ car $r < 0$. Nous venons de montrer que si $r$ était strictement négatif, alors la boucle se serait arrêtée à l'itération précédente, ce qui est absurde. Donc nécessairement $r \geq 0$.

Résumons, le programme s'arrête avec $0 \leq r < b$ et la propriété $a = bq + r$ est vérifiée à chaque itération, ces deux propriétés nous démontrent que l'algorithme effectue bien la division de $a$ par $b$.

### 2.2.3 Un autre exemple

Déterminons un algorithme qui calcule le produit de deux entiers $a$ et $b$ avec $a$ positif ou nul sans effectuer de multiplication.

```
p ⟵ 0
m ⟵ 0
tant que m < a
|  p ⟵ p + b
|  m ⟵ m + 1
fin tant que
```

- Montrons que le programme se termine. La séquence formée par les valeurs prises par $m$ au fil des itérations est strictement croissante, il y a donc un moment où $m \geq a$. Donc l'algorithme se termine.
- Considérons comme invariant de boucle $p = m.b$, montrons par récurrence qu'il est vérifiée.
  - $m$ est $p$ sont initialisés à 0, on a bien $0 = 0.b$.
  - montrons maintenant que la propriété est conservée. Supposons qu'au début d'une itération donnée, on ait $p = m.b$. Soient $p'$ et $m'$ les valeurs de ces variables à la fin de cette itération, montrons que l'on a $p' = m'.b$. On a les relations $p' = p + b$ et $m' = m + 1$. Donc $p' = p + b = m.b + b = (m + 1).b = m'.b$. La propriété est donc conservée par chaque itération de l'algorithme.
- La première de valeur de $m$ qui nous fera sortir de la boucle est $a$, donc après la dernière itération on aura $p = mb = ab$. Ce programme place donc dans $p$ le produit de $a$ par $b$.

#### Exercice 1 - Exponentiation

Ecrire un algorithme de calcul de $b^n$ où $n$ est un entier positif ou nul. Prouver sa terminaison et sa validité.

#### Exercice 2 - Plus grand élément

Ecrire un algorithme de recherche du plus grand élément dans un tableau. Prouvez sa terminaison et sa validité. On rappelle que $m$ est le plus grand élément du tableau $T = [T_1, \ldots, T_n]$ si les deux conditions suivantes sont vérifiées :

- $\forall i \in \{1, \ldots, n\}, m \geq T_i$
- $\exists j \in \{1, \ldots, n\}, m = T_j$

#### Exercice 3 - Tri par insertion

Nous allons écrire et démontrer la validité de l'algorithme du tri par insertion. Nous rappelons que $T = [T_1, \ldots, T_n]$ est trié si $\forall i \in \{1, \ldots, n-1\}, T_i \leq T_{i+1}$. Etant donné le programme suivant :

```
Procédure insere(T, k)
    i ⟵ k  tant que i > 0 ∧ T[i − 1] > T[i]
        echanger(T[i − 1], T[i])
        i ⟵ i − 1
    fin tant que
FIN
```

Prouver que si on passe en paramètre à *insere* un tableau $T$ dont les $k-1$ premiers éléments sont triés, alors à la fin de son exécution, les $k$ premiers éléments dont triés. Etant donné le programme suivant :

```
Procédure triInsertion(T[n])
    pour i ∈ {2, . . . , n}
        insere(T, i)
    fin pour
FIN
```

Démontrer la terminaison et la validité de *triInsertion*.

Voici le corrigé de la première question : L'algorithme se termine bien car les valeurs prises par $i$ forment une séquence de nombres entiers strictement décroissante. Si l'on ne sort pas de la boucle avec $T_{i-1} \leq T_i$, on aura forcément $i = 1$ après la dernière itération. Prenons comme invariant de boucle la conjonction des conditions suivantes :

- La tranche $T_1, \ldots, T_{i-1}$ est triée
- La tranche $T_i, \ldots, T_k$ est triée
- $i < k \Rightarrow T_{i-1} \leq T_{i+1}$

Prouvons cette propriété par récurrence :

- A la première itération, on a $i = k$. Comme $T_1, \ldots, T_i$ correspond à la tranche $T_1, \ldots, T_{k-1}$ donc par hypothèse cette tranche est triée. La tranche $T_i, \ldots, T_k$ est réduite au seul élément $T_k$, elle est donc nécessairement triée. Comme $\neg(i < k)$, la troisième condition est trivialement vérifiée.
- Supposons qu'au début d'une itération arbitraire, les trois propriétés soient vérifiées. Soient $i'$ la valeur de $i$ et $T'$ le tableau $T$ à la fin de cette itération. Montrons alors que
  - La tranche $T'_1, \ldots, T'_{i'-1}$ est triée
  - La tranche $T'_{i'}, \ldots, T'_k$ est triée
  - $i' < k \Rightarrow T'_{i'-1} \leq T'_{i'+1}$

  D'une part, on a $i > 1$ et $T_{i-1} > T_i$. Par ailleurs, $i' = i - 1$, $T'_{i-1} = T_i$ et $T'_i = T_{i-1}$. Tous les autres éléments de $T$ et de $T'$ coïncident. Montrons les trois propriétés :
  - Par hypothèse de récurrence, la tranche $T_1, \ldots, T_{i-1}$ est triée, donc la tranche $T_1, \ldots, T_{i-1}$ que l'on obtient en ne comptabilisant pas le dernier élément est aussi triée. Comme les éléments de $T$ et $T'$ d'indices différents de $i$ et $i + 1$ sont les mêmes et que $i - 1 = i'$, alors $T'_1, \ldots, T'_{i'}$ est triée.
  - Par hypothèse de réccurrence $T_i, \ldots, T_k$ est triée, en ne considérant pas le premier élément, on a $T_{i+1}, \ldots, T_k$, comme les éléments de cette tranche coïncident avec $T'$, alors $T'_{i'+2}, \ldots, T'_k$ est triée. Par ailleurs, on a $T'_{i'} = T_i$ et $T'_{i'+1} = T_{i-1}$, comme $T_{i-1} > T_i$, alors $T'_{i'} < T'_{i'+1}$. Si $i < k$, alors par hypothèse de récurrence, on a $T_{i-1} \leq T_{i+1}$, donc $T'_{i'+1} \leq T'_{i'+2}$. Dans ce cas, on a $T'_{i'} < T'_{i'+1} \leq T'_{i'+2}$ avec $T'_{i'+2}, \ldots, T'_k$ triée. Donc $T'_{i'}, \ldots, T'_k$ est triée. Si $i = k$, alors la tranche $T'_{i'}, \ldots, T'_k$ est réduite aux deux éléments $T'_{i'} < T'_{i'+1}$, elle est donc triée.
  - Quelle que soit la valeur de $i$, on a nécessairement $i' < k$, il faut donc montrer que $T'_{i'-1} \leq T'_{i'+1}$. Or $T'_{i'-1} = T_{i-2}$ et $T'_{i'+1} = T_{i-1}$. Par hypothèse de récurrence, $T_1, \ldots, T_{i-1}$ est triée, donc on a $T_{i-2} \leq T_{i-1}$, d'où $T'_{i'-1} \leq T'_{i'+1}$.

Après la denière itération, si $i = 1$, alors la première tranche ne contient aucun élément et la deuxième condition de l'invariant de boucle permet de conclure que toute la tranche $T_1, \ldots, T_k$ est triée. Si par contre, on sort de la boucle parce que $T_{i-1} \leq T_i$, alors comme, d'après l'invariant de boucle, $T_1, \ldots, T_{i-1}$ et $T_i, \ldots, T_k$ sont triées, alors $T_1, \ldots, T_k$ est triée.

### Exercice 4 - Tri à bulle

Ecrire et démontrer la validité de l'algorithme du tri à bulle.

## 2.3 Complexité

On détermine la complexité d'un programme itératif en comptant le nombre d'itérations. Lorsque deux boucles sont imbriquées, on est amené à compter pour chaque itération de la boucle principale le nombre d'itérations de la boucle imbriquée et à les additionner. Par exemple,

```
pour i ∈ {1, ..., n − 1}
    pour j ∈ {i + 1, ..., n}
        (* opération en O(1) *)
    fin pour
fin pour
```

A l'itération $i$ de l'algorithme, la boucle imbriquée effectue $n-(i+1)+1 = n-i$ itérations. On détermine donc le nombre total d'itérations en sommant les $n-i$, donc avec

$$(n-1)+(n-2)+\ldots+(n-(n-1)) = \sum_{i=1}^{n-1}(n-i) = \frac{((n-1)-1+1)(n-1+n-(n-1))}{2} = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

**Exercice 5 - Sommations**

Simplifier les sommes suivantes :

- $\sum_{i=1}^{n+2} i$
- $\sum_{i=0}^{n} i+1$
- $e^{\sum_{i=1}^{n} ln(i)}$
- $\sum_{i=1}^{n}(3i-2)$
- $\sum_{i=1}^{n}\sum_{j=i}^{n-1} 1$
- $\sum_{i=1}^{n}(3(\sum_{j=i}^{n-1} 1)-2)$
- $\sum_{i=5}^{n+3}(i-1)$
- $\sum_{i=1}^{n}(2n-3i)$

**Exercice 6 - Tris quadratiques**

Calculer les complexités du tri par insertion et du tri à bulle.

**Morceaux choisis**

**Exercice 7 - Polynômes**

Nous représenterons un polynôme $P(x) = p_0x^n + p_1x^{n-1} + \ldots + p_{n-1}x + p_n = \sum_{i=0}^{n} p_i x^{n-i}$ de degré $n$ à l'aide d'un tableau $[p_0, p_1, \ldots, p_{n-1}, p_n]$ à $n+1$ éléments. Notez bien que ce tableau est indicé à partir de 0.

1. Que fait la fonction suivante ?

```
Fonction mystery([p_0, ..., p_n], x)
    somme ⟵ 0
    pour i ∈ {0, ..., n}
        terme ⟵ 1
        j ⟵ 1
        tant que j ≤ n − i
            terme ⟵ terme × x
            j ⟵ j + 1
        fin tant que
        terme ⟵ terme × p_i
        somme ⟵ somme + terme
    fin pour
    retourner somme
FIN
```

2. Quelle est la complexité de *mystery* ?
3. Écrire la procédure $derive([p_0, \ldots, p_n])$ remplaçant $p$ par sa dérivée.
4. Prouver que $\sum_{i=0}^{k+1} p_i x^{k+1-i} = x \left( \sum_{i=0}^{k} p_i x^{k-i} \right) + p_{k+1}$
5. Déduire de la relation ci-dessus une fonction $evalue([p_0, \ldots, p_n], x)$ prenant $P$ et $x$ en paramètres, et retournant $P(x)$.
6. Prouver la validité de la fonction d'évaluation, vous utiliserez les questions précédentes.
7. Quelle est la complexité de *evalue* ?

**Exercice 8 - Tranche minimale**

Etant donné un tableau $T[n]$, un tranche est une suite d'éléments contigüs de $T$. Une tranche $[T_i, \ldots, T_j]$ est entièrement déterminée par l'indice $i$ de début et l'indice $j$ de fin. La valeur de la tranche $[T_i, \ldots, T_j]$ est donnée par la somme

$$V_i^j = \sum_{k=i}^{j} T_k$$

Nous souhaitons mettre au point un algorithme de recherhe de la tranche minimale.

1. Ecrivez un algorithme construit avec trois boucles imbriquées. Calculez sa complexité.
2. Utilisez le fait que $V_i^{j+1} = V_i^j + T[j+1]$ pour écrire un algorithme construit avec deux boucles imbriquées. Calculez sa complexité.
3. Soient $M_i^j$ la valeur de la tranche minimale incluse dans la tranche $[T_i, \ldots, T_j]$, et soit $\hat{M_i^j}$ la tranche minimale de $[T_i, \ldots, T_j]$ contenant $T_j$. Donnez une relation entre $M_i^{j+1}$, $M_i^j$ et $\hat{M_i^j}$.
4. En déduire un algorithme de recherche de la tranche minimale construit avec une seule boucle. Déterminer sa complexité.

# Chapitre 3

# Récursivité

## 3.1 Sous-programmes récursifs

Un sous-programme **récursif** est un sous-programme qui s'appelle lui-même. Par exemple :

**Procédure** *countDown*($n$)
  **si** $n \geq 0$ **alors**
    *afficher*($n$)
    *countDown*($n-1$)
  **fin**
**FIN**

Ce sous-programme affiche $n$ si celui-ci est positif ou nul, puis se rappelle en passant $n-1$ en paramètre. Il va se rappeler jusqu'à ce que le paramètre prenne la valeur $-1$, affichant ainsi un compte à rebours dont la dernière valeur affichée sera 0. Lorsque qu'un sous-programme se rappelle, on dit qu'il effectue un **appel récursif**. Pour observer ce que cela fait, considérez le programme suivant :

```
#include<stdio.h>

void countDown(int n)
{
  if (n >= 0)
    {
      printf("%d\n", n);
      countDown(n-1);
    }
}

int main()
{
  countDown(10);
  return 0;
}
```

Il affiche

```
10
9
8
7
6
5
4
```

```
3
2
1
0
```

Observons bien la procédure *countdown* :

- La première chose faite est le test $n \geq 0$, il faut toujours tester la **condition d'arrêt** avant de faire quoi que ce soit. Si cela n'est pas fait correctement, le sous-programme peut boucler indéfiniment.
- Dans l'appel récursif, la valeur $n-1$, et non pas $n$, est passée en paramètre. Si vous ne faites pas décroître la valeur de $n$, le sous-programme se comportera de façon identique à chaque appel récursif, donc bouclera.
- L'appel récursif est fait après l'affichage de $n$, si vous permutez l'affichage et l'appel récursif, les nombres seront affichés dans l'ordre croissant. un sous-programme est **récursif terminal** si la dernière instruction exécutée est un appel récursif, ce qui est le cas ici.

Beaucoup de sous-programmes sont beaucoup plus faciles à implémenter de façon récursive. Un sous-programme utilisant des boucles mais pas la récursivité est dit **itératif**. Comme la récursivité permet d'éviter de faire des boucles, on oppose souvent la mode récursif au mode itératif. On exposera divers algorithmes s'implémentant naturellement de façon récursive, notamment le tri fusion, et les algorithmes dans les listes et les arbres. Pour le moment nous nous limiterons au calcul de valeurs définies par récurrence.

### 3.1.1 Factorielle

Soit $n$ un nombre entier positif ou nul, le nombre $n!$, dit "factorielle $n$", est défini comme suit :

$$n! = 1 \times 2 \times \ldots \times n = \prod_{i=1}^{n} i$$

avec le cas particulier $0! = 1$. On observe que

$$n! = [1 \times \ldots \times (n-1)] \times n = [(n-1)!] \times n$$

Cette relation nous permet de réduire le calcul de la factorielle d'un nombre $n$ au calcul de la factorielle de $n-1$. En utilisant cette relation on obtient un algorithme récursif calculant la factorielle d'un nombre $n$.

**Fonction** $factorielle(n)$
  **si** $n = 0$ **alors**
    **retourner** 1
  **sinon**
    **retourner** $n \times factorielle(n-1)$
  **fin**
**FIN**

- Observez bien que la condition d'arrêt est testée dès le début de l'exécution de l'algorithme, et qu'elle permet de calculer la valeur 0!
- Dans le cas général, le calcul $n!$ se fait en passant par le calcul de $(n-1)!$. Comme la valeur passée en paramètre dans l'appel récursif est $n-1$ et est strictement inférieure à $n$, alors la séquence des valeurs passées en paramètre au fil des appels récursifs

  $$n \longrightarrow n-1 \longrightarrow \ldots \longrightarrow 1 \longrightarrow 0$$

  est strictement décroissante et converge vers 0. Il est très important de vérifier que quelques soient les valeurs passées en paramètre, la séquence de valeurs formée par les appels récursifs converge vers une valeur satisfaisant la condition d'arrêt, 0 dans le cas présent.
- On remarque par ailleurs que cet algorithme n'est pas récursif terminal, en effet l'appel récursif précède une multiplication par $n$, la dernière instruction n'est donc pas un appel récursif.

Les doutes des plus sceptiques seront, je l'espère apaisés par la vue du sous-programme suivant :

```
long factorielle(long n)
{
  if (n == 0)
```

```
        return 1;
    return n * factorielle(n - 1);
}
```

**Exercice 1 - Opérations arithmétiques entières**

Considérez l'algorithme suivant :

**Fonction** $addition(a, b)$
  **si** $a = 0$ **alors**
    **retourner** $b$
  **sinon**
    **retourner** $1 + addition(a - 1, b)$
  **fin**
**FIN**

1. Quelle relation arithmétique utilise-t-il ?
2. Modifiez-le pour qu'il puisse additionner des nombres négatifs.
3. Ecrivez un algorithme récursif soustrayant deux nombres de signes quelconques.
4. Ecrivez un algorithme récursif multipliant deux nombres de signes quelconques.
5. Ecrivez un algorithme récursif divisant deux nombres de signes quelconques.
6. Ecrivez un algorithme récursif calculant $a^b$, avec des valeurs de $b$ entières positives ou nulles.

**Exercice 2 - La fonction** *mystery*

Considérons la fonction suivante :

**Fonction** $mystery(n)$
  **si** $n = 0$ **alors**
    **retourner** 2
  **sinon**
    **retourner** $[mystery(n - 1)]^2$
  **fin**
**FIN**

1. Que fait cette fonction ?
2. Prouvez par récurrence votre conjecture de la question précédente.
3. Écrire une fonction récursive $mysteryBis(a, b, n)$ sur le même modèle que *mystery* retournant $a^{(b^n)}$. Vous utiliserez la fonction $puissance(x, m)$ qui retourne $x^m$.
4. Prouvez la validité de $mysteryBis$.

## 3.2 Sous-programmes récursifs terminaux

Bon nombre de sous-programmes non terminaux peuvent être réécrits de façon terminale. Cela se fait par l'ajout d'un paramètre, appelé **accumulateur**. La forme d'un programme récursif terminal est la suivante :

**Fonction** $f(\ldots, r)$
  **si** $conditionDArret$ **alors**
    **retourner** $r$
  **sinon**
    $instructions$
  **fin**
**FIN**

Il faut faire en sorte qu'une fois la condition d'arrêt vérifiée le paramètre $r$ contienne le résultat.

### 3.2.1 Factorielle

Étudions un exemple,

```
Fonction factorielleT(n, r)
  | si n = 0 alors
  |  | retourner r
  | sinon
  |  | retourner factorielleT(n − 1, r × n)
  | fin
FIN
```

Tout d'abord, on remarque que $factorielleT$ est bien une fonction récursive terminale. Ensuite observons que si l'on invoque $factorielleT(n, 1)$, on obtient la séquence suivante :

$$(n, 1) \longrightarrow (n-1, n) \longrightarrow (n-2, n \times (n-1)) \longrightarrow \dots$$

$$\dots \longrightarrow (i, \prod_{k=i+1}^{n} k) \longrightarrow \dots$$

$$\dots \longrightarrow (2, \prod_{k=3}^{n} k) \longrightarrow (1, \prod_{k=2}^{n} k) \longrightarrow (0, \prod_{k=1}^{n} k)$$

Au moment où la condition d'arrêt est vérifiée, le paramètre $r$ contient la valeur $\prod_{k=1}^{n} k$, à savoir $n!$. On se sert de l'accumulateur pour fabriquer le résultat. Démontrons par récurrence sur $n$ que $factorielleT(n, r) = r(n!)$ :

- Tout d'abord, on constate que $factorielleT(0, r) = r$
- Supposons que $factorielleT(n-1, r')$ retourne la valeur $(n-1)! \times r'$, donc $factorielleT(n-1, n \times r) = (n-1)!(r \times n) = r(n!)$. D'où $factorielleT(n, r) = factorielleT(n-1, n \times r) = r(n!)$

On en déduit qu'en posant $r = 1$, on a $factorielleT(n, 1) = n!$ On encapsule $factorielleT$ dans le sous-programme $factorielle$ que l'on redéfinit de la sorte :

```
Fonction factorielle(n)
  | retourner factorielleT(n, 1)
FIN
```

### 3.2.2 Exponentiation

Essayons de créer une fonction récursive terminale calculant $b^n$, comme cette valeur s'obtient par des multiplications successives, l'accumulateur $a$ est une valeur par lequel le résultat sera multiplié. Nous souhaitons donc mettre un point une fonction $puissanceT(a, b, n)$ qui retourne $a(b^n)$. On obtiendra $b^n$ en initialisant $a$ à 1. On détermine une relation de récurrence entre $b^n$ et $b^{n-1}$ de la sorte : $b^n = b.b^{n-1}$, donc $ab^n = (ab)b^{n-1}$, à savoir $puissanceT(a \times b, b, n-1)$. Allons-y,

```
Fonction puissanceT(a, b, n)
  | si n = 0 alors
  |  | retourner a
  | sinon
  |  | retourner puissanceT(a × b, b, n − 1)
  | fin
FIN
```

Prouvons par récurrence sur $n$ que $puissanceT(a, b, n) = ab^n$.

- On a bien $puissanceT(a, b, 0) = a = ab^0$
- Supposons que $puissanceT(a', b, n-1) = a'b^{n-1}$, donc $puissanceT(ab, b, n-1) = (ab)b^{n-1} = ab^n$. D'où $puissanceT(a, b, n) = puissanceT(ab, b, n-1) = ab^n$.

On calcule donc $b^n$ en invoquant $puissanceT(1, b, n)$. On redéfinit $puissance(b, n)$ de la sorte :

**Fonction** $puissance(b, n)$
  **retourner** $puissanceT(1, b, n)$
**FIN**

Vous avez des doutes ? Traduisez donc ces sous-programmes en C en observez ce qui se passe...

**Exercice 3 - Récursion terminale**

Reprennez tous les sous-programmes de l'exercice précédent et rédigez-les sous forme récursive terminale. Vous prouverez par récurrence la validité de chacune d'elle.

### 3.2.3 Itérateurs

**Exercice 4 - Introduction aux itérateurs**

Etant donné une fonction $f$, on note $f^n$ la composition de $f$ par elle-même $n$ fois. Par exemple : $f^3 = f \circ f \circ f$, si $f : x \mapsto x + 1$, alors $f^3(x) = f(f(f(x))) = x + 3$. Par convention, on a $f^0 = id$, avec $id : x \mapsto x$.

1. Si $f : x \mapsto x + 1$, démontrez par récurrence que $f^n : x \mapsto x + n$
2. Si $f : x \mapsto 2x$, démontrez par récurrence que $f^n : x \mapsto 2^n x$
3. Si $f : x \mapsto x^2$, démontrez par récurrence que $f^n : x \mapsto x^{(}2^n)$
4. Ecrire une fonction récursive $applyF$ non terminale prenant en paramètre une fonction $f$, un nombre $n$ et un argument $x$, et retournant $f^n(x)$. Ce type de fonction sera appelé un **itérateur**.
5. Ecrire une version récursive terminale de $applyF$.
6. Soit $f : (a, b) \mapsto (a + b, b)$, montrez par récurrence que $f^n : (0, b) \mapsto (bn, b)$
7. Ecrire une fonction $f$ prenant en paramètre le couple $(a, b)$ et retournant $(a + b, b)$.
8. Ecrire la fonction *multiplie* en utilisant $f$ et $applyF$.

**Exercice 5 - Applications des itérateurs**

1. Réfléchir à un moyen d'implémenter les itérateurs en C, n'oubliez pas que la fonction itérée peut prendre et retourner plusieurs paramètres.
2. Pour des raisons de lisibilité du code, nous utiliserons des structures pour représenter les couples. Cela a pour inconvénient un manque de généricité de la fonction $f$. Utiliser des tableaux offrirait une souplesse optimale mais alourdirait considérablement le code du fait des allocations et libérations de la mémoire nécessaires. Implémentez en C la fonction *multiplie* de l'exercice précédent, vous utiliserez bien évidemment un itérateur.
3. Ecrivez la fonction $factorielle$ à l'aide d'un itérateur, démontrez sa validité et traduisez-là en C.
4. Ecrivez la fonction *puissance* à l'aide d'un itérateur, démontrez sa validité et traduisez-là en C.

**Exercice 6 - Itérateurs et aspirine**

Dans les questions précédentes, l'itérateur faisait décroître $n$ de 1 à chaque appel récursif. Soit $f$ la fonction à itérer et $x$ la valeur en 0, nous pouvons définir un tel itérateur $\mathcal{I}$ de la sorte :
- $\mathcal{I}(f, n, x) = f(\mathcal{I}(f, n - 1, x))$
- $\mathcal{I}(f, 0, x) = x$

Nous étendons une telle fonction en ajoutant en paramètre une fonction de décrément $\delta$, on obtient :
- $\mathcal{I}(\delta, f, n, x) = f(n, \mathcal{I}(\delta, f, \delta(n), x))$
- $\mathcal{I}(\delta, f, 0, x) = x$.

1. Implémentez la fonction $\mathcal{I}(\delta, f, n, x)$ en en utilisant le prototype suivant `unsigned long applyFD(int (*delta)(int k), unsigned long (*f)(couple), int n, unsigned long x)`.
2. Ecrivez une fonction $f$ telle que $\mathcal{I}(\delta, f, n, b) = b^n$, vous utiliserez la relation de récurrence $b^n = b.b^{n-1}$ et $b^0 = 1$ et poserez $\delta : x \mapsto x - 1$.

3. Prouvez que $\mathcal{I}(\delta, f, n, b) = b^n$
4. Ecrivez en C la fonction `unsigned long slowPuissanceIT(unsigned long b, unsigned long n)`, vous utiliserez `applyFD` et traduirez en C les fonctions $f$ et $\delta$ définies dans la question préccédente.
5. Ecrivez deux fonctions $f$ et $\delta$ telles que $\mathcal{I}(\delta, f, n, 1) = n!$
6. Prouvez que $\mathcal{I}(\delta, f, n, 1) = n!$
7. Ecrivez en C la fonction `unsigned long fatorielleIT(unsigned long b, unsigned long n)`, vous utiliserez `applyFD`.
8. Ecrivez une fonction récursive $fastPuissance(b, n)$ calculant $b^n$ sans utiliser d'itérateur, vous utiliserez le fait que $b^n = (b^2)^{\frac{n}{2}}$ si $n$ est pair et $b^n = b(b^2)^{\lfloor \frac{n}{2} \rfloor}$ sinon.
9. Traduisez-là en C en utilisant le prototype suivant `unsigned long fastPuissance(unsigned long b, unsigned long n)`.
10. Majorez le nombre d'appels récursifs en fonction de $n$, est-ce performant ?
11. Ecrivez la fonction $f$ et la fonction $\delta$ de sorte que $\mathcal{I}(\delta, f, n, b) = b^n$. Notez le fait que $f$ prend le couple $(n, b)$ en paramètre.
12. Prouvez par récurrence la validité de $\mathcal{I}(\delta, f, n, b) = b^n$.
13. Ecrivez en C la fonction `unsigned long fastPuissanceIT(unsigned long b, unsigned long n)`, vous utiliserez la fonction `applyFD`.

**Exercice 7 - Itérateurs et nombres de Fibonacci**

1. Soit $\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$, un vecteur de deux nombres de fibonacci consécutifs. Que donne le produit $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$ ?
2. Prouvez par récurrence que $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$.
3. On calcule donc le $n$-ème nombre de Fibonacci en mettant préalablement la matrice $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ à la puissance $n$. Nous allons utiliser pour ce faire la même règle que pour $fastPuissance$. Ecrivez une fonction $multMat$ prenant en paramètre deux quadruplets $(x_{11}, x_{12}, x_{21}, x_{22})$ et $(y_{11}, y_{12}, y_{21}, y_{22})$ représentant deux matrices $X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$ et $Y = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$. $multMat$ retourne un quadruplet représentant le produit matriciel $XY$.
4. Définir une fonction $matPow(X, n)$ mettant un matrice $X$, à la puissance $n$. Vous utiliserez $\mathcal{I}(\delta, multMat, n, X)$.
5. Ecrire une fonction $multMatVec$ retournant le produit d'une matrice, représentée par un quadruplet, et d'un vecteur, représenté par un couple.
6. Ecrire en C la fonction `quadruplet applyFDQ(int (*delta)(int k), unsigned long (*f)(int, quadruplet), int n, quadruplet x)`, vous utiliserez le même modèle mathématique, mais en transmettant comme deuxième paramètre à $f$ un quadruplet.
7. Ecrire la fonction $fastFibonacci(n)$, retournant le $n$-ième nombre de fibonacci.
8. Quelle est la complexité de $fastFibonacci$ ?
9. Implémenter les fonctions :
   (a) `quadruplet multMat(quadruplet x, quadruplet y)`
   (b) `quadruplet applyFDQ(unsigned long (*delta)(unsigned long), quadruplet (*f)(unsigned long, quadruplet), unsigned long n, quadruplet x`
   (c) `quadruplet matPow(quadruplet x, unsigned long n)`
   (d) `couple multMatVec(quadruplet x, couple y)`
   (e) `unsigned long fastFibonacciIT(unsigned long l)`.

## 3.3 Listes chaînées

### 3.3.1 Notations et conventions

La récursivité est particulièrement adaptée lorsque l'on souhaite manipuler des listes chaînées. Nous utiliserons pour ce faire les fonctions suivantes :

1. $premier(l)$ retourne la **donnée** se trouvant dans le premier élément de la liste chaînée $l$.
2. $suivants(l)$ retourne un **pointeur** $l'$ vers le deuxième élément de $l$, cette fonction ne produit aucun effet de bord et ne modifie pas le chaînage.
3. $ajoute(z, l)$ ajoute la donnée $z$ au début de la liste $l$ et retourne un **pointeur** vers la liste obtenue, elle modifie le chaînage.
4. $estVide(l)$ retourne vrai si et seulement si la liste $l$ ne contient aucun élément.
5. $listeVide()$ retourne une constante que nous identifierons à la liste vide, c'est-à-dire ne contenant aucun élément.

Vous prendrez soin de ne pas confondre les listes et les données. Ces dernières sont génériques, vous pouvez placer dans une liste des données de n'importe quel type, même d'autres listes... Nous noterons mathématiquement les listes entre des crochets, par exemple $[1, 4, 3, 7]$. La liste vide, retournée par la fonction $listeVide()$ sera notée $[]$. Vous pourrez créer une liste à un élément par exemple en écrivant

$$ajoute(4, listeVide())$$

Cette instruction crée la liste $[4]$. Pour créer une liste à deux éléments, vous utiliserez le même principe :

$$ajoute(3, ajoute(4, listeVide()))$$

Cette instruction crée la liste $[3, 4]$. Les éléments de la liste sont indicés à partir de 1. Nous ne gérerons pas la mémoire, on considérera qu'un garbage collector s'en occupera.

### 3.3.2 Rédaction

Les règles de la récursivité sont les mêmes quand on manipule des listes chaînées, on teste le cas de base en premier : la liste vide, et on fait en sorte que la séquence formée par les valeurs passées en paramètre au file des appels récursifs converge vers la liste vide. Par exemple,

```
Procédure sommeListe(l)
    si estVide(l) alors
        retourner 0
    sinon
        retourner premier(l) + sommeListe(suivants(l))
    fin
FIN
```

Cette fonction retourne la somme des éléments de $l$.

#### Exercice 8 - Prise en main

Ecrire les fonctions suivantes en pseudo-code :
- $listPrint(l)$, affiche les éléments de la liste $l$.
- $listDeleteByIndex(l, k)$ retourne une liste contenant les éléments de la liste $l$ sauf celui d'indice $k$.
- $listDeleteByValue(l, a)$, retourne une liste contenant les éléments de $l$ sauf le premier rencontré à avoir la valeur $a$.
- $listDeleteAllByValue(l, a)$, retourne une liste contenant les éléments de $l$ sauf tous les éléments ayant la valeur $a$.
- $listOcurrences(l, a)$, retourne une liste contenant les indices de toutes les ocurrences de $a$ dans $l$.
- $listIndexesUnsorted(l, i)$ retourne une liste contenant tous les éléments de la liste $l$ dont l'indice se trouve dans la liste $i$. $i$ n'est pas supposée triée.

–
– $listIndexesSorted(l, i)$ retourne une liste contenant tous les éléments de la liste $l$ dont l'indice se trouve dans la liste $i$. $i$ est supposée triée.
– $listDeleteByIndexes(l, i)$ retourne une liste contenant tous les éléments de $l$ dont l'indice se trouve dans $i$, $i$ est supposée triée.

**Exercice 9 - Tris**

Ecrire les fonctions suivantes en pseudo-code :
– $listInsert(l, a)$, insère la donnée $k$ dans la liste $l$, retourne un pointeur vers le premier élément de la liste.
– $listInsertionSort(l)$, retourne les éléments de $l$ triés par la méthode du tri par insertion.
– $listSplit(l)$, retourne un couple de listes $(m, n)$ dans lesquelles ont été repartis de façons arbitraire les éléments de $l$ et telle que le nombre de données de $m$ et $n$ diffère au plus de 1.
– $listMerge(l1, l2)$, retourne une liste $l$ triée contenant les éléments de $l1$ et de $l2$. $l1$ et $l2$ sont supposées triées.
– $listFusionSort(l)$ retourne une liste contenant les éléments de $l$ triés avec la méthode du tri fusion.

## 3.4 Exemple de calcul de complexité d'un algorithme récursif

### 3.4.1 L'algorithme

La complexité d'un algorithme récursif s'exprime en règle générale par l'intermédiaire d'une relation de récurrence. Par exemple, l'affichage des éléments d'une liste chaînée, donné par l'algorithme suivant :

```
Procédure afficheListe(l)
    si ¬estVide(l) alors
        afficher(premier(l))
        afficheListe(suivants(l))
    fin
FIN
```

Soit $u_n$ le nombre d'instructions élémentaires exécutées par $afficheListe$ si la liste $l$ comporte $n$ maillons. Si $n = 0$, l'algorithme s'exécute en temps constant. Sinon, l'appel récursif nécessite $u_{n-1}$ opérations, les autres opérations (au nombre de $k$) s'exécutent en temps constant. On définit donc $u$ comme suit : $u_n = u_{n-1} + k$, comment exprimer $u_n$ en fonction de $u_0$, de $n$ et de $k$ ?

### 3.4.2 Résolution exacte

Il apparaît que $u$ est une suite arithmétique de raison $k$ et de premier terme $u_0$, donc $u_n = u_0 + kn$. Comme $\frac{u_0 + kn}{n} \longrightarrow k$, alors $u_n \in \mathcal{O}(n)$, donc $afficheListe$ est de complexité linéaire. On remarque que, dans ce type de suite, $u_o$ et $k$ sont des valeurs n'ayant pas de conséquence sur le caractère linéaire de la complexité de ce type d'algorithme. On se permettra donc, dans la plupart des cas, de les remplacer par 1. Nous aurons donc les fois suivantes $u_0 = 1$ et $u_n = u_{n-1} + 1$, le résultat, en terme de complexité sera le même.

### 3.4.3 Vérification par récurrence

Il est fréquent que l'on ne parvienne pas à exprimer $u_n$ en fonction de $u_0$ et de $n$, certaines récurrences sont très difficile à résoudre. Une autre façon de classifier une suite $u_n$ dans un $\mathcal{O}$ est de conjecturer ce $\mathcal{O}$, puis de le démontrer par récurrence. Par exemple, démontrons par récurrence que $u_n \in \mathcal{O}(n)$. Si c'est le cas, supposons que $u_n$ est majorée par une suite linéaire, c'est à dire de la forme $an + b$. Choisissons $b$ tel que $b \geq u_0 = 1$ et $a > 1$.
– L'hypothèse sur $b$ implique que $a(0) + b \geq u_0$.
– Supposons que si $an + b \geq u_n$, alors montrons que $a(n + 1) + b \geq u_{n+1}$. On a $u_{n+1} = u_n + 1 \leq an + b + 1 \leq an + b + a = a(n + 1) + b$.

On montre aisément que $a$ et $b$ choisis conformément aux conditions précédant la démonstration par récurrence détermine une suite $an+b$ qui majore $u_n$. On remarque que $an+b$ croît de façon linéaire, en effet $\frac{an+b}{n} \longrightarrow a$, donc on a $an+b \in \mathcal{O}(n)$. Comme $u_n$ est majorée par une suite linéaire, $u_n \in \mathcal{O}(n)$.

Prouvons pour le sport que si $u_n \leq v_n$ et $v_n \in \mathcal{O}(w_n)$, alors $u_n \in \mathcal{O}(w_n)$. Comme $v_n \in \mathcal{O}(w_n)$, alors $\exists k, \exists N, \forall n > N, v_n \leq k.w_n$, comme $u_n \leq v_n$, alors pour les mêmes valeurs $k$ et $N$, $u_n \leq v_n \leq k.w_n$ et de ce fait $u_n \leq k.w_n$.

Deux ensembles de méthodes ressortent dans les exemples :

- La résolution exacte, elle nécessite le connaissance de bon nombre de méthodes mathématiques, bien qu'élégante, elle devient vite très compliquée.
- La conjecture d'une majoration suivie d'une preuve par récurrence de cette majoration. La preuve est en règle générale plus simple à mettre en oeuvre, la partie la plus difficile est de conjecturer une majoration valide et la moins grossière possible.

### 3.4.4 Problèmes

**Exercice 10 - Tours de Hanoï**

On dispose de $n$ disques numérotés de 1 à $n$. Les disques peuvent être empilés sur des socles à partir du moment où tout disque repose sur un disque de numéro plus élevé. On dispose de trois socles $S_1$, $S_2$ et $S_3$, les $n$ disques sont empilés sur le socle de gauche $S_1$, ceux du milieu et de droite $S_2$ et $S_3$ sont vides. Le but est de déplacer toute la pile de disques sur le socle de droite, sachant qu'on ne peut déplacer les disques que un par un, et qu'il ne faut qu'à aucun moment, un disque numéroté $k$ se trouve sur un disque numéroté $m$ avec $m < k$ (i.e un disque plus petit). Par exemple, si $n = 3$, on a

```
  1
  2
  3
------------
  2
  3       1
------------
  3   2   1
------------
      1
  3   2
------------
      1
      2   3
------------
  1   2   3
------------
          2
  1       3
------------
          1
          2
          3
------------
```

On dispose d'un sous-programme $deplaceDisque(S_i, S_j)$ qui enlève le disque se trouvant au sommet de la pile $S_i$ et la place au sommet de la pile $S_j$.

1. Trouver un algorithme récursif $deplaceDisques(n, S_{from}, S_{via}, S_{to})$ permettant de déplacer $n$ disques du socle $S_{from}$ au socle $S_{to}$ en utilisant $S_{via}$ comme socle intermédiaire.
2. Codez en C un programme qui affiche chaque étape du déplacement des $n$ disques, que remarquez-vous lorsque l'on prend des grandes valeurs de $n$ ?

3. Déterminer une relation de récurrence permettant de dénombrer le nombre d'étapes nécessaires pour déplacer $n$ disques. Résolvez-là.
4. Est-il envisageable de déplacer 100 disques en un temps raisonnable ?

**Exercice 11 - Suite de Fibonacci**

1. Ecrire une fonction récursive `unsigned long fibo(unsigned long l)` calculant le $n$-ème nombre de fibonacci $F_n$ défini par récurrence de la sorte : $F_n = F_{n-1} + F_{n-2}$ avec $F_0 = 0$ et $F_1 = 1$.
2. Soit $u_n$ le nombre d'appels récursifs nécessaires pour calculer $F_n$. Prouvez par récurrence que $u_n \geq F_n$.
3. Exprimer $F_n$ en fonction de $n$ en résolvant la récurrence linéaire.
4. Déterminer la complexité de `fibo` en passant par une borne asymptotique inférieure.
5. On considère la fonction $\phi : (a, b) \mapsto (a + b, a)$. Prouvez que $\phi(F_{n-1}, F_{n-2}) = (F_n, F_{n-1})$.
6. On note $\phi^k$ la composition de $k$ fonctions $\phi$. Par exemple, $\phi^2 = \phi \circ \phi$ et $\phi^2(a, b) = \phi(\phi(a, b)) = \phi(a+b, a) = (2a+b, a+b)$. Par convention, $\phi^0$ est la fonction identité, notée $id$. Prouver par récurrence que $(F_{n+1}, F_n) = \phi^n(F_1, F_0))$.
7. Implémentez $\phi$ avec le sous-programme `void phi(unsigned long* l1, unsigned long* l2)`, notez que les deux variables `l1` et `l2` sont passées en paramètre par référence.
8. Ecrivez le corps du sous-programme `void applyF(void (*f)(unsigned long*, unsigned long*), int n, unsigned long* l1, unsigned long* l2)`, `applyF` applique `n` fois la fonction `f`, en utilisant comme paramètre lors du premier appel les variables pointées par `l1` et `l2`.
9. Utilisez `applyF` et `phi` pour coder `unsigned long fastFibo(unsigned long l)`
10. Majorer la complexité de `applyF` lorsqu'on lui passe le sous-programme `phi` en paramètre, prouver ce résultat par récurrence.
11. Quelle est la complexité de `fastFibo` ? Porte-t-elle bien son nom ?

**Exercice 12 - Pgcd**

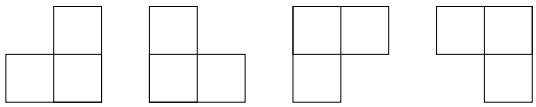
Le plus grand commun diviseur de deux entiers relatifs $a$ et $b$, noté $pgcd(a, b)$, est défini par la relation de récurrence $pgcd(a, b) = pgcd(b, a \ mod \ b)$, avec $pgcd(a, 0) = a$.

1. Ecrire une fonction récursive calculant le *pgcd* de deux entiers relatifs.
2. Ecrire une fonction itérative retournant le $i$-ème nombre de fibonacci.
3. Ecrire les deux sous-programmes `unsigned long pgcd(unsigned long a, unsigned long b)` et `unsigned long fibonacci(unsigned long l)`
4. Tester la fonction *pgcd* avec des nombres de fibonacci consécutifs. Qu'observez-vous ?
5. Si on s'intéresse au nombre d'appels récursifs nécessaires pour calculer un *pgcd*, quel est le pire des cas ?
6. Quel est la valeur du quotient $\frac{F_{n+1}}{F_n}$ ?
7. En déduire que les entrées de la forme ($F_{n+1}, F_n$ forment les pires cas.
8. Combien d'itérations sont nécesaires pour calculer $pgcd(F_{n+1}, F_n)$ ?
9. Prouver que $F_n$ est majoré par une suite géométrique.
10. Prouver que le nombre d'appels récursifs est majoré par un logarithme de base $\phi = \frac{1+\sqrt{5}}{2}$.
11. Quelle est la complexité de *pgcd*, est-ce efficace ?

**Exercice 13 - Pavage avec des L**

Soit $n$ un nombre positif fixé. On considère une grille carrée de $2^n \times 2^n$ cases. Un case arbitraire est noircie, toutes les autres sont libres. Le problème consiste à paver toute la grille sauf la case noircie avec des pièces en forme de L. Ces pièces occupent trois cases :

1. résolvez le problème pour $n = 0$.



2. résolvez le problème pour tout $n \geq 0$, vous montrerez que si l'on est capable de résoudre ce problème pour une valeur $n$ donnée, alors il est possible de le résoudre pour une valeur $n+1$.
3. programmez cet algorithme en C, remplissez chaque case avec un caractère. Vous représenterez une pièce en affectant le même caractère à toutes les cases occupées par cette pièce. Par exemple (avec $n = 4$),

```
@ @ ? ? ; ; : : + + * * & & % %
@ > > ? ; 9 9 : + ) ) * & $ $ %
A > B B < < 9 = , ) - - ' ' $ (
A A B 8 8 < = = , , - # # ' ( (
E E D 8 J J I I 0 0 / # 5 5 4 4
E C D D J H H I 0 . / / ! 5 3 4
F C C G K H L L 1 . . 2 6 3 3 7
F F G G K K L " 1 1 2 2 6 6 7 7
U U T T P P O " " j i i e e d d
U S S T P N O O j j h i e c c d
V S W W Q N N R k h h l f f c g
V V W M Q Q R R k k l l b f g g
Z Z Y M M _ ^ ^ o o n b b t s s
Z X Y Y _ _ ] ^ o m n n t t r s
[ X X \ ` ] ] a p m m q u r r v
[ [ \ \ ` ` a a p p q q u u v v
```

Vous remarquez que la case noircie est celle avec le point d'exclamation. Représentez la matrice en mémoire de sorte à faciliter l'implémentation récursive de l'algorithme. C'est à dire en indicant de la façon suivante (si $n = 3$) ;

```
 0  1  4  5 16 18 20 21
 2  3  6  7 17 19 22 23
 8  9 12 13 24 25 28 29
10 11 14 15 26 27 30 31
32 33 36 37 48 49 52 53
34 35 38 39 50 51 54 55
40 41 44 45 56 57 60 61
42 43 46 47 58 59 62 63
```

### 3.4.5 L'heure du code

Maintenant c'est l'heure du code. Etes-vous capable de programmer les algorithmes étudiés dans les exercices précédents ? C'est ce que nous allons voir...

#### Exercice 14 - Echauffement : représentation des complexes

Ecrivez les fonctions du fichier `complexe.c` correspondant au fichier `complexe.h` ci-dessous :

```
#ifndef COMPLEXE_H
#define COMPLEXE_H

#include "linkedList.h"

/*
  Type complexe
 */

typedef struct
{
```

```
  /*
    Partie reelle
  */
  double re;

  /*
    partie imaginaire
  */
  double im;

  /*
    module
   */
  double mod;

  /*
    argument, nul si le nombre est 0 + 0i
   */
  double arg;
}complexe, *pComplexe;


double reComplexe(complexe c);

/*
  Retourne la partie imaginaire
 */
double imComplexe(complexe c);

/*
  Retourne le module
 */
double modComplexe(complexe c);

/*
  Retourne l'argument
 */
double argComplexe(complexe c);

/*
  Affiche c sous forme exponentielle
 */
void printExpComplexe(complexe c);

/*
  Affiche l'affixe de c
 */
void printAffComplexe(complexe c);

/*
  Cree un complexe a partir de son affixe
 */
complexe makeAffComplexe(double re, double im);

/*
  Cree un complexe a partir de sa forme exponentielle
 */
complexe makeExpComplexe(double mod, double arg);

/*
  Retourne le conjugue de c
 */
complexe conjugueComplexe(complexe c);

/*
  Soustrait c1 et c2
 */
complexe subComplexe(complexe c1, complexe c2);

/*
  Additionne c1 et c2
 */
complexe addComplexe(complexe c1, complexe c2);

/*
  Multiplie c1 et c2
 */
complexe multComplexe(complexe c1, complexe c2);

/*
  Multiplie  c par le reel d
```

```
 */
complexe multReelComplexe(double d, complexe c);

/*
  Divise c2 par c1
 */
complexe divComplexe(complexe c1, complexe c2);

/*
  Eleve c a la puissance n
 */
complexe puissComplexe(complexe c, unsigned long n);

/*
  Retourne un pointeur vers une copie du complexe c.
*/
pComplexe copyComplexe(complexe c);

#endif
```

### Exercice 15 - Une bibliothèque de listes chaînées

Ecrivez les fonctions du fichier `linkedList.c` correspondant au fichier `linkedList.h` ci-dessous :

```
#ifndef LINKED_LIST_H

#define LINKED_LIST_H

/*
  Maillon de la liste chainee
*/

typedef struct l
{
  /*
    pointeur vers la donnee
  */
  void* data;

  /*
    pointeur vers le maillon suivant
  */
  struct l* next;
} link;

/*
  Type encapsulant la liste chainee
*/

typedef struct
{
  /*
    nombre d'elements de la liste chainee
   */
  int size;

  /*
    pointeur vers le premier elemtn de la liste chainee
   */
  link* first;

  /*
    pointeur vers le dernier element de la liste chainee
   */
  link* last;
} linkedList;


/*
 Cree une liste chainee ne contenant aucun element.
*/

linkedList* linkedListCreate();

/*
  Ajoute l'element x dans a la fin de la liste chainee l
*/

void linkedListAppend(linkedList* l, void* x);
```

```c
/*
  Ajoute le maillon lk a la fin de la liste l
*/

void linkedListAppendLink(linkedList* l, link* lk);

/*
  Ajoute l'element x dans au debut de la liste chainee l
*/

void linkedListPush(linkedList* l, void* x);

/*
  Ajoute le maillon lk au debut de la liste l
*/

void linkedListPushLink(linkedList* l, link* lk);

/*
  Retourne la taille de la liste l
*/

int linkedListGetSize(linkedList* l);

/*
  Retourne un pointeur vers le premier maillon de
  la liste chainee l, NULL si l est vide.
*/

link* linkedListGetFirst(linkedList* l);

/*
  Retourne un pointeur vers le premier maillon de l,
  enleve ce maillon de la liste l. Retourne NULL
  si l est vide.
*/

link* linkedListUnlinkFirst(linkedList* l);

/*
  Retourne une liste chainee contenant toutes les images
  des donnees par la fonction f. En posant f = id, on obtient
  une fonction de copie.
*/

linkedList* linkedListMap(linkedList* l, void* (*f)(void*));

/*
  Concatene les deux listes begin et end, la deuxieme est ajoutee
  a la fin de la premiere.
*/

void linkedListConcat(linkedList* begin, linkedList* end);

/*
  Applique la fonction f a toutes les donnees de la liste l.
  Passer une fonction d'affichage permet par exemple d'afficher
  les donnees de chaque maillon.
*/

void linkedListApply(linkedList* l, void (*f)(void*));

/*
  Desalloue l et tous ses maillons, applique le destructeur fr
  a la donnee de chaque maillon.
*/

void linkedListDestroy(linkedList*, void (*fr)(void*));

#endif
```

### Exercice 16 - Tri par insertion

Nous allons implémenter le tri par insertion d'une liste chaînée : Vous utiliserez les sous-programmes suivants et n'utiliserez aucune boucle, seule la récursivité est autorisée.

- `link* insert(link* list, link* item, int (*estInf)(void*, void* ))`, insère dans la liste `list` triée le maillon `item`. On a `estInf(i, j)` si et seulement si la clé de `i` est strictement inférieure à celle de `j`.

- `link* triInsertion(link* list, int (*estInf)(void*, void*))`, trie par insertion la liste `list` en utilisant la fonction de comparaison `estInf`.
- `void linkedListSort(linkedList* l, int (*estInf)(void*, void*))`, trie la liste `l` en modifiant le chaînage, vous n'oublierez pas de mettre à jour `l->first` et `l->last`.
- Identifez le pire des cas et observez la façon dont le temps d'exécution croît avec le nombre de maillons. Est-ce que cela corrobore les conclusions théoriques ?

**Exercice 17 - Tri fusion**

Le tri par fusion se fait en coupant un tableau en deux sous-tableaux de tailles égales à un élément près, en triant récursivement les deux sous-tableaux puis en interclassant leurs éléments. Ecrire en C une fonction de tri fusion de listes chaînées. Vous prendrez garde à ne pas recopier les maillons, et à seulement modifier le chaînage. Vous utiliserez les sous-programmes énoncés ci-après. Notez bien le type des paramètres.

- `void split(linkedList* source, linkedList* dest1, linkedList* dest2)`, répartit les maillons de `source` dans `dest1` et `dest2`.
- `void merge(linkedList* source1, linkedList* source2, linkedList* dest, int (*inf) (void*, void*))`, interclasse les listes `source1` et `source2` et place le résultat dans la liste `dest`. Le fonction pointée par $f$ retourne 1 si la clé premier paramètre est inférieur à la clé du deuxième paramètre, 0 sinon.
- `void triFusion(linkedList* l, int (*inf)(void*, void*))`, tri la liste `l` en modifiant le chaînage.
- Comparez son temps d'exécution à celui du tri par sélection. Déterminez sa complexité. Est-ce un algorithme de tri efficace ?

**Exercice 18 - Transformée discrète de Fourier**

Nous allons étudier la transformée discrète de Fourier et la façon dont on peut la calculer efficacement. Etant donné un vecteur $a = (a_0, \ldots, a_{n-1})$, la transformée discrète de Fourier de $a$ est un vecteur $b = (b_0, \ldots, b_{n-1})$, noté $b = TF(a)$ dont le $p$-ème élément est

$$b_p = \sum_{k=0}^{n-1} a_k (e^{\frac{p}{n} 2\pi i})^k$$

1. Commençons par étudier les racines complexes de 1. Vous vous doutez probablement 1 a deux racines carrées $-1$ et 1. Vérifiez que 1, $\frac{-1}{2} + \frac{\sqrt{3}}{2} i$ et $\frac{-1}{2} - \frac{\sqrt{3}}{2} i$ sont 3 racines cubiques de 1.
2. Vérifiez que 1, $1 + i$, $-1$ et $1 - i$ sont 4 racines quatrièmes de 1.
3. Vérifiez que $W_n = \{e^{\frac{1}{n} 2\pi i + \frac{k}{n} 2i\pi} | k \in \{0, \ldots, n-1\}\}$ est l'ensemble des racines $n$-èmes de 1.
4. Posons $\omega_n = e^{\frac{1}{n} 2\pi i}$, véfifiez que $W_n = \{\omega_n^k | k \in \{0, \ldots, n-1\}\}$.
5. Prouvez que pour tous $n$ et $k$, $\omega_n^{k+n} = \omega_n^k$.

On considère l'algorithme

```
Fonction A(a, x, n)
    r ⟵ 0
    pour i ∈ {0, ..., n − 1}
        y ⟵ 1
        pour j ∈ {1, ..., i}
            y ⟵ y × x
        fin pour
        r ⟵ r + a_i × y
    fin pour
    retourner r
FIN
```

1. Soit $P(x) = a_0 + \sum_{i=1}^{n-1} a_i x_i$ un polynome de degré $n - 1$. Que vaut l'algorithme $A(P, x, n)$ ?

2. Quelle est la complexité de $A$ ?
3. La méthode de Horner est basée sur la relation suivante $P(x) = a_0 + x(a_1 + x(a_2 + x(\ldots)))$. Ecrivez l'algorithme récursif *horner* qui calcule $P(x)$ en utilisant cette relation.
4. Quelle est la complexité de *horner* ?
5. Soit $P(x) = a_0 + \sum_{i=1}^{n-1} a_i x_i$ un polynôme de degré $n-1$ et $b = TF(a)$. Montrez que $b_p = P(w_n^p)$.
6. Comment calculer $TF(a)$ en utilisant le sous-programme *horner* ? Ecrivez le sous-programme $TF$.
7. On note $TF^{-1}$ la transformée inverse de Fourier, nous admettrons que si $a = TF^{-1}(b)$, alors

$$a_q = \frac{1}{n} \sum_{k=0}^{n-1} b_k \omega_n^{-kq}$$

   Ecrivez le sous-programme $TF^{-1}$.
8. Quelle sont les complexités de $TF$ et de $TF^{-1}$ ?
9. Prouvez que pour tous $n$ et $k$, $\omega_{2n}^{2k} = \omega_n^k$.
10. Supposons que $n$ est pair, donc qu'il existe $m$ tel que $n = 2m$. Soit $a^{[0]} = (a_0, a_2, \ldots, a_{2m})$, $a^{[1]} = (a_1, a_3, \ldots, a_{2m+1})$, $b^{[0]} = TF(a^{[0]})$ et $b^{[1]} = TF(a^{[1]})$. Soit $q \in \{0, \ldots, m-1\}$, montrez que

$$b_q = b_q^{[0]} + \omega_n^q b_q^{[1]}$$

    puis que

$$b_{q+m} = b_q^{[0]} + \omega_n^{q+m} b_q^{[1]}$$

11. Ecrire le sous-programme $FFT$ qui calcule la transformée discrète de Fourier d'un vecteur de taille $n = 2^k$ en utilisant la relation de récurrence ci-dessus.
12. Quelle est la complexité de $FFT$ ?
13. Supposons que $n$ est pair, donc qu'il existe $m$ tel que $n = 2m$. Soit $b^{[0]} = (b_0, b_2, \ldots, b_{2m})$, $b^{[1]} = (b_1, b_3, \ldots, b_{2m+1})$, $y^{[0]} = TF^{-1}(b^{[0]})$ et $y^{[1]} = TF^{-1}(b^{[1]})$. Soit $p \in \{0, \ldots, m-1\}$ et $y = TF^{-1}(b)$. Montrez que

$$y_p = \frac{1}{2}(y_p^{[0]} + \omega_n^{-p} y_p^{[1]})$$

    puis que

$$y_{p+m} = \frac{1}{2}(y_p^{[0]} + \omega_n^{-(p+m)} y_p^{[1]})$$

14. Ecrire le sous-programme $FFT^{-1}$ qui calcule la transformée inverse de Fourier d'un vecteur de taille $n = 2^k$ en utilisant la relation de récurrence ci-dessus.
15. Quelle est la complexité de $FFT^{-1}$ ?
16. Ecrivez les fonctions du fichier `fourier.c` correspondant au fichier `fourier.h` ci-dessous.

```
#ifndef FOURIER_H
#define FOURIER_H

#include"linkedList.h"
#include"complexe.h"

/*
  Retourne la transformee discrete de Fourier du vecteur l.
  Algorithme iteratif en O(n^2).
 */
linkedList* transformeeFourierIt(linkedList* l);

/*
  Retourne la transformee discrete de Fourier inverse
  du vecteur l. Algorithme iteratif en O(n^2).
 */
linkedList* transformeeInverseFourierIt(linkedList* l);
```

```
/*
  Retourne la transformee discrete de Fourier du vecteur l, de taille
  n = 2^m. Algorithme recursif en O(n log n).
 */
linkedList* FFT(linkedList* l);

/*
  Retourne la transformee discrete de Fourier inverse
  du vecteur l de taille n = 2^m. Algorithme recursif en
  O(n log n).
 */
linkedList* inverseFFT(linkedList* l);

#endif
```

Vous utiliserez les fonctions auxiliaires suivantes :

1. `linkedList* uniteRacinesComplexe(complexe c, unsigned long n)`, retourne les `n` racines `n`-ème de 1.
2. `complexe horner(link* p, complexe x)`, retourne l'image de `x` par le polynôme `p`.

### Exercice 19 - Représentation des polynômes

Nous souhaitons implémenter une bibliothèque de représentation des fonctiosn polynômes. La plupart des opérations sont relativement simples à rédiger. La plus délicate est la multiplication. Prenons un exemple, pour multiplier $P(x) = 1 + x + 2^2$ et $Q(x) = 1 - x + x^2$, il y a deux façons de procéder.

1. En multipliant "bêtement" : $P(x)Q(X) = (1+x+2^2)(1-x+x^2) = 1(1-x+x^2)+x[(1-x+x^2)+2x(1-x+x^2)] = (1-x+x^2)+x[1+x-x^2+2x^3] = 1-x+x^2+x+x^2-x^3+2x^4 = 1+2x^2-x^3+2x^4$.
2. Une autre méthode consiste à passer par une interpolation. Le polynôme $PQ$ sera évidemment de degré 4, donc en ayant 5 points par lesquels passa la courbe de $PQ$, il est possible d'interpoler ses coefficients. Pour trouver ces points, il suffit d'évaluer $P$ et $Q$ en 5 points, par exemple $\{1, 2, 3, 4, 5\}$, On obtient 2 vecteurs

   $$\{(1, P(1)), (2, P(2)), (3, P(3)), (4, P(4)), (5, P(5))\}$$

   et

   $$\{(1, Q(1)), (2, Q(2)), (3, Q(3)), (4, Q(4)), (5, Q(5))\}$$

   Cela nous donne 5 points par lesquels passe la courbe de $PQ$ :

   $$\{(1, P(1)Q(1)), (2, P(2)Q(2)), (3, P(3)Q(3)), (4, P(4)Q(4)), (5, P(5)Q(5))\}$$

   Il suffit ensuite d'effectuer une interpolation des coefficients $a, b, c, d$ et $e$ de la courbe de $PQ$, cela se fait en résolvant le système d'équations

   $$\{a1^4 + b1^3 + c1^2 + d1^1 + e = P(1)Q(1), a2^4 + b2^3 + c2^2 + d2^1 + e = P(2)Q(2), \ldots\}$$

Chacune de ces méthoes requiert $\mathcal{O}(n^2)$ opérations. Il est cependant possible d'affiner la deuxième en choisissant judicieusement les points servant à évaluer puis à interpoler. En prenant $W_n$ comme ensemble de points, on ramène les évaluations de ces points à des transformées de Fourier et l'interpolation à une transformation inverse de Fourier. Cela permet de multiplier des polynômes en $\mathcal{O}(n \log_2 n)$. Voyez [2] pour un exposé davantage détaillé. Ecrivez les fonctions du fichier `polynomes.c` correspondant au fichier `polynomes.h` ci-dessous. Vous prendrez soin de vous assurer que la multiplication à l'aide de la transformée de Fourier est plus rapide que la version pour boeufs.

```
#ifndef POLYNOME_H
#define POLYNOME_H

#include"linkedList.h"
#include"complexe.h"
#include"fourier.h"

/*
  Nous representerons un polynome avec une liste chainee contenant
  les coefficients du polynome par ordre de degre croissant.
*/

/*
Retourne un polynome vide.
 */
```

```
linkedList* makePolynome();

/*
  Retourne une copie de l, copie aussi les coefficients.
*/
linkedList* copyPolynome(linkedList* l);

/*
  Detruit le polynome l et ses coefficients.
 */
void destroyPolynome(linkedList* l);

/*
  Affiche polynome le plus proprement possible.
*/
void printPolynome(linkedList* polynome);

/*
  Retourne le degre de l.
 */
int degreePolynome(linkedList *l);

/*
  Ajoute un monome de coefficient coeff a la fin du polynome, faisant
  ainsi croutre son degre de 1.
*/
void appendCoeffPolynome(linkedList* polynome, double coeff);

/*
  Cree et retourne le polynome de degre n-1 dont les
  coefficients sont passes dans le tableau d.
 */
linkedList* makeArrayPolynome(double* d, int n);

/*
  Retourne le polynome nul.
*/
linkedList* zeroPolynome();

/*
  Retourne le polynome unite.
*/
linkedList* unitPolynome();

/*
  Ajoute la constante d au polynome l.
*/
void addRealPolynome(linkedList* l, double d);

/*
  Multiplie le polynome l par le reel d.
*/
void multRealPolynome(linkedList* l, double d);

/*
  Multiplie le polynome l par X^n
*/
void multXNPolynome(linkedList* l, int n);

/*
  Multiplie le polynome l par coeff*X^exp
*/
void multMonomePolynome(linkedList* l, double coeff, int exp);

/*
  Additione les deux polynomes a et b, retourne cette somme.
 */
linkedList* addPolynome(linkedList* a, linkedList* b);

/*
  Multiplie a et b avec la recurrence
  (a_0 + a_1X + ... + a_nX^n)Q(X)
  = a_0 * Q(X) + X * (a_1 + a_2 X...+ a_n X^(n-1)) * Q(X)
*/
linkedList* slowMultPolynome(linkedList* a, linkedList* b);

/*
  Multiple a et b en passant par une transformee de fourier.
 */
linkedList* multPolynome(linkedList* a, linkedList* b);
```

```
/*
  Evalue le polynome l en x avec la methode de Horner.
 */
double evaluatePolynome(linkedList *l, double x);

#endif
```

# Chapitre 4

# Arbres

## 4.1 Définitions par induction

On remarque que l'ensemble des arbres binaires est défini par induction. C'est-à-dire à partir d'un ensemble d'**atomes** et de **règles**. Les atomes sont les éléments de l'ensemble qui sont indécomposables, par exemple l'arbre vide $\emptyset$. Les règles permettent à partir d'éléments d'un ensemble de créer d'autres éléments. Par exemple, si $A$ et $B$ sont des arbres binaires, et $r$ un noeud, alors le triplet $(A, r, B)$ est un arbre binaire.

### 4.1.1 Définition

On définit un ensemble $E$ par induction en deux étapes :
- **La base** La base est une liste d'éléments de $E$ qui sont des atomes. On les notes $\mathcal{A}$ et on la propriété $\mathcal{A} \subset E$.
- **L'induction** L'induction est un ensemble de règles permettant de créer un élément de $E$ avec d'autres éléments de $E$. Chaque règle est une fonction $f : E^k \longrightarrow E$ telle que si $(E_1, \ldots, E_k)$ est un $k$-uplet d'éléments de $E$, alors $f(E_1, \ldots, E_k) \in E$.

### 4.1.2 Exemple

Considérons l'ensemble $\mathbb{N}$ des entiers naturels, on le définit de la sorte :
- **Base** la seul atome de $\mathbb{N}$ est 0. Donc On les note $\mathcal{A} = \{0\}$.
- **Induction** On prend comme unique règle la fonction $s : x \mapsto x + 1$ appelée successeur. Pour tout élément $n$ de $\mathbb{N}$, on a $s(n) \in N$.

On conviendra donc que $\mathbb{N}$ est l'ensembles de tous les éléments faisant soit partie de $\mathcal{A}$, soit pouvant être obtenu par application d'un nombre fini de règles. Par exemple, $0 \in \mathbb{N}$, car $0 \in \mathcal{A}$, $4 \in \mathbb{N}$, car $0 \in \mathcal{A}$ et $4 = s(s(s(s(0)))) \in \mathbb{N}$ (vous remarquez que 4 s'obtient par application d'un nombre fini de règles).

### 4.1.3 Expressions arithmétiques

Une expression arithmérique totalement parenthésée (EATP) se définit de la sorte,
- **Base** Tout nombre $x \in \mathbb{R}$ est une EATP.
- **Induction** Soit $E, E'$ deux EATPs, alors $(E + E')$, $(E - E')$, $(E \times E')$ et $(E/E')$ sont des EATPs.

Une juxtaposition de symboles $X$ est une EATP s'il est possible de générer $X$ en applicant un nombre fini de fois les règles précédentes. Si l'on souhaite définir formellement une EATP, il convient de les représenter de façon ensembliste. On redéfinit donc l'ensemble $\mathcal{E}$ des EATPs de la sorte :
- **Base** $\mathbb{R} \subset \mathcal{E}$
- **Induction** $\forall E, E' \in \mathcal{E}$, $+(E, E'), -(E, E'), \times(E, E'), /(E, E') \in \mathcal{E}$.

Cette définition met en correspondance des expressions comme $(3 + (4 \times 5))$ avec des ensembles de la forme $+(3, \times(4, 5))$, plus faciles à manier dans des algorithmes. Naturellement, on représentera toute expression arithmérique par un arbre binaire. Les feuilles représenteront les atomes de nos expressions et les noeuds représenteront les règles.

### 4.1.4 Application aux arbres binaires

Soit $f$ une fonction prenant an argument un arbre binaire, la façon la plus claire et la précise de définir une telle fonction est souvent de la caractériser par induction. Par exemple

**Définition 4.1.1** *On note $h(A)$ la hauteur de l'arbre $A$. On la définit par induction :*
- $h(\emptyset) = -1$
- $h((A_g, r, A_d)) = 1 + \max(h(A_g), h(A_d))$

**Définition 4.1.2** *On note $|A|$ le nombre de noeuds d'un arbre $A$. On le définit par induction :*
- $|\emptyset| = 0$
- $|(A, r, B)| = 1 + |A| + |B|$

## 4.2 Preuves par induction

### 4.2.1 Lien avec les preuves par récurrence

Si on veut montrer que tout élément de $\mathbb{N}$ vérifie une propriété, par exemple $\forall n \in \mathbb{N}$, $n(n+1)$ est pair, on va construire par induction l'ensemble $E \subset \mathbb{N}$ des $n \in \mathbb{N}$ tels que $n(n+1)$ est pair. Si les atomes de $\mathbb{N}$ et de $E$ sont les mêmes et que l'on parvient à construire $\mathbb{N}$ et $E$ avec les mêmes règles, alors $\mathbb{N} = E$. Nous avons deux étapes à effectuer pour le vérifier :
- **Base** la seul atome de $\mathbb{N}$ est 0. Comme $0 \times 1 = 0$ est pair, alors $0 \in E$.
- **Induction** La seule règle que nous avons à examiner est $s : x \mapsto x + 1$. Prenons un élément arbitraire de $E$ ($E$ n'est pas vide puisqu'il contient 0), notons-le $n$. Montrons que $s(n) \in E$. On a $s(n) = n + 1$, donc $s(n) \in E$ si $(n+1)(n+1)$ est pair. Or $(n+1)(n+2) = (n+1)n + (n+1)2$, comme $n \in E$, alors $n(n+1)$ pair. Par ailleurs $2(n+1)$ est évidemment pair. Comme la somme de deux nombres pairs est paire, alors $s(n) \in E$.

Nous venons de montrer que $\mathbb{N}$ peut être construit de la même façon que $E$, donc $\mathbb{N} \subset E$ (et par conséquent $E = \mathbb{N}$). Comme $E$ est l'ensemble des entiers $n$ tels que $n(n+1)$ est pair, alors pour tout élément $n$ de $\mathbb{N}$, $n(n+1)$ est pair. Ce que nous venons de faire est en fait une preuve par récurrence. La raisonnement par récurrence est un cas particulier du raisonnement par induction.

### 4.2.2 Application aux EATPs

On veut démontrer que toute EATP contient au moins autant de parenthèses que d'opérateurs. Procédons en ayant à l'esprit la définition d'une EATP :
- **Base** Les seules EATPs atomiques les expressions réduites à un réel, dont ne contenant ni parenthèse, ni opérateur.
- **Induction** Les quatre règles à considérer sont de la même forme : $E = (E_g o E_d)$ où $o$ est un opérateur binaire, $E_g, E_d$ deux EATPs, et $E$ une EATP non atomique pour laquelle on cherche à démontrer la propriété. Par hypothèse d'induction, $E_g$ et $E_d$ contiennent au moins autant de parenthèses que d'opérateurs. Et dans l'expression $(E_g o E_d)$ on ajoute plus de parenthèses que d'opérateurs, donc $E$ contient au moins autant de parenthèses que d'opérateurs.

#### Exercice 1 - Opérandes et opérateurs

Démontrez par induction que dans toute EATP, il y a une opérande de plus qu'il n'y a d'opérateurs.

## 4.3 Algorithmique dans les arbres

On définit une bijection *cle* entre les noeuds de $A$ et un ensemble $(E, >)$ totalement ordonné. On note $cle(x)$ la clé du noeud $x$. En général, on prend $E = \mathbb{N}$.

Nous représenterons chaque noeud d'un arbre binaire avec une structure $ABR$ contenant les champs suivants :
- *cle*
- $g$
- $d$

Si $a$ est une variable de type AB alors $a.cle$ est la clé de la racine de l'arbre $a$, $a.d$ est un pointeur vers le sous-arbre droit, $a.g$ est un pointeur vers le sous-arbre gauche. Si un pointeur ne pointe vers aucune valeur, on dira qu'il prend la valeur *null*. On fera une allocation dynamique en utilisant la fonction $AB(f_g, c, f_d)$ pour créer un noeud de clé $c$ et de sous-arbre gauche (resp. droit) $f_g$ (resp. $f_d$) de type AB.

**Exercice 2 - Nombre de noeuds**

Ecrire une fonction retournant le nombre de noeuds formant un arbre $A$.

**Exercice 3 - Hauteur**

Ecrire une fonction retournant la hauteur d'un arbre $A$.

**Exercice 4 - Profondeur**

Ecrire une fonction retournant la profondeur d'un noeud $x$ dans un arbre $A$.

## 4.4 Arbres $n$-naires

**Définition 4.4.1** *$A$ est un arbre $n$-aire de si une des conditions suivantes est vérifiée*
- *$A$ est un arbre vide, on le note abusivement $A = \emptyset$.*
- *$A = (d, l)$ où*
  - *$d$ est la racine de $A$*
  - *$l$ est un $k$-uplet ($k \leq n$) d'arbres $n$-aires.*

On remarque qu'un arbre 1-aire est une liste chaînée. Nous représenterons un arbre $n$-aire avec une structure ARBRE, dont le constructeur est de même nom.

# Chapitre 5

# Files de priorité

Une file de priorité est une structure de donnée permettant de stocker un ensemble d'éléments munis d'une clé d'ordre total, et dans laquelle on s'intéresse aux opérations suivantes :
- Extraction du plus petit élément
- Ajout d'un élément
- Suppression du plus petit élément

## 5.1 Implémentations naïves

Une première idée serait d'implémenter une file de priorité avec un tableau non trié. Dans ce cas, on a :
- Extraction du plus petit élément en $\mathcal{O}(n)$
- Ajout d'un élément en $\mathcal{O}(1)$
- Suppression du plus petit élément $\mathcal{O}(n)$

Cette solution est peu satisfaisante. Avec un tableau trié, une liste chaînée triée ou non, on obtient des résultats peu satisfaisants. Nous allons devoir méditer sur des solutions permettant d'obtenir des temps de calcul au moins inférieurs à $\mathcal{O}(log_2 n)$.

**Exercice 1 - Tableau non trié**

Justifier les résultats de complexité.

**Exercice 2 - Tableau trié**

Déterminer les complexités des trois opérations dans le cas où le tableau est trié.

**Exercice 3 - Liste chaînée non triée**

Déterminer les complexités des trois opérations dans le cas où on utilise une liste chaînée non triée.

**Exercice 4 - Liste chaînée triée**

Déterminer les complexités des trois opérations dans le cas où on utilise une liste chaînée triée.

## 5.2 Tas

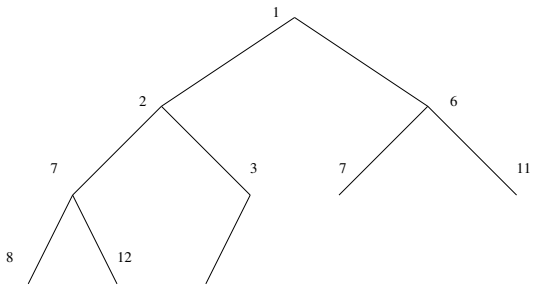
### 5.2.1 Définition

Un tas est un arbre binaire vérifiant les propriétés suivantes :

1. Tous les niveaux sont remplis sauf éventuellement le dernier.
2. Dans le dernier niveau, les noeuds sont disposés le plus à gauche possible.

3. Tous noeud possède une clé inférieure à celle de ses deux fils.

Par exemple,




Vous remarquez que les trois premiers niveaux sont remplis et que dans le quatrième niveau, les noeuds disposés le plus à gauche possible. De même, on constate que chaque noeud a une clé inférieure à celle de ses fils.

**Exercice 5 - Définition**

Représenter un tas contenant les éléments suivants : $\{3, 87, -1, 0, 3, 9, 54, -3\}$. Vous disposerez les éléments à votre convenance du moment que l'arbre considéré est bien un tas.

**Exercice 6 - Hauteur**

Majorer la hauteur d'un tas contenant $n$ éléments.

### 5.2.2 Extraction du minumum

Vous remarquez que le plus petit élément est à la racine. Par conséquent l'extraction est trivialement en temps constant.

### 5.2.3 Insertion

Pour insérer un élément dans un tas, on choisit l'unique emplacement qui permet de conserver les deux premières propriétés ? Ensuite on réequilibre le tas. Le soin de cette procédure vous est laissé en exercice.

**Exercice 7 - Ajout**

Décrire la procédure permettant de déterminer où insérer un nouvel élément dans un tas.

### 5.2.4 Suppression du minimum

Supprimer la dernière feuille du tas est simple, c'est la seule operation qui conserve les deux premières propriétés. La solution que nous préconiserons sera de permuter la racine avec le dernier élément. Il suffira ensuite d'équiliber le tas autour de la racine.

**Exercice 8 - Equilibrage d'un tas**

Décrire une façon d'équilibrer un tas après insertion d'un nouvel élément. Donner un invariant de boucle. Est-ce que le programme se termine ?

### 5.2.5 Implémentation

En règle générale, on implémente un tas avec un tableau. Les éléments sont disposés dans l'ordre d'un parcours en largeur de la gauche vers la droite.

**Exercice 9 - Indices**

Nous indicerons les tableaux à partir de 1. Déterminez une relation de récurrence entre l'indice d'un noeud et les indices de ses voisins (père, fils gauche, fils droit).

**Exercice 10 - Performances**

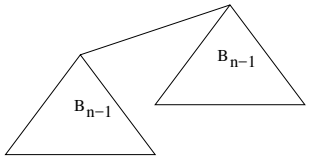
Faites le point sur les temps d'exécution des diverses opérations. Est-ce efficace ?

## 5.3 Tas binomial

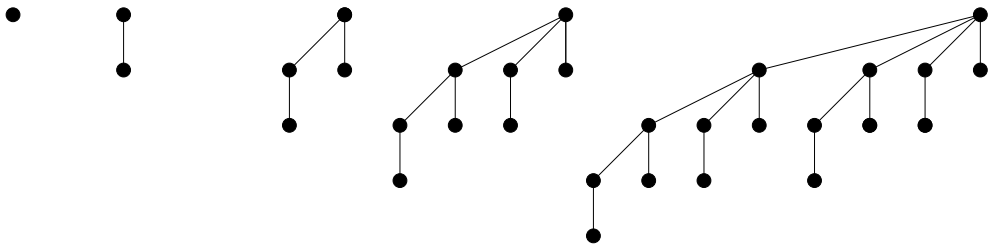
Ajoutons maintenant une opération : la fusion. Etant données deux fils de priorités, comment les fusionner efficacement ?

### 5.3.1 Arbre binomial

Un arbre binomial d'ordre $n$ (noté $B_n$) est défini par induction de la façon suivante :
- $B_0$ est réduit à un seul noeud.
- $B_n$ est construit de la façon suivante :




Voici par exemple $B_0$, $B_1$, $B_2$, $B_3$ et $B_4$ :



**Exercice 11 - Représentation graphique**

Représenter les 5 premiers arbres binomiaux.

**Exercice 12 - Propriétés**

Prouver par induction les propriétés suivantes :

1. $B_n$ contient $2^n$ noeuds.
2. $B_n$ est de hauteur $n$.
3. La racine de $B_n$ a $n$ fils.
4. Les $n$ fils de la racine de $B_n$ sont $B_0$, $B_1$, ..., $B_{n-1}$.
5. Il y a $\mathcal{C}_n^k$ noeuds de profondeur $k$ dans $B_n$.

### 5.3.2 Fusion

Etant donnés deux arbres binomiaux, on les fusionne en faisant de l'un un fils de la racine de l'autre. Cette opération (jeu d'adresses) se fait en temps constant.

### 5.3.3 Tas binomial

Un tas binomial est une liste d'arbres binomiaux vérifiant les propriétés suivantes :
- Dans chaque arbre, tout noeud est de clé inférieure à ses fils.
- Tous les arbres sont d'ordres distincts.

### 5.3.4 Implémentation

Les tas binomiaux ne sont pas triviaux à implémenter. On représente un tas binomial par une liste chaînée d'arbre binomiaux. Chaque arbre binomial est représenté par un ordre, une clé, et un pointeur vers la liste chaînée de ses fils. Le premier des fils est l'arbre d'ordre le plus élevé.

**Exercice 13 - Implémentation**

Représenter graphiquement la représentation en mémoire d'un tas binomial.

**Exercice 14 - Représentation binaire de $n$**

Montrer qu'il est toujours possible de répartir $n$ clés dans des arbres binomiaux d'ordres distincts. Majorer le nombre d'arbres binomiaux nécessaires en fonction de $n$.

### 5.3.5 Extraction du minimum

Comme tous les arbres sont équilibrés, le plus petit élément du tas est nécessairement une racine. Il suffit donc d'examiner tous les racines.

**Exercice 15 - Complexité**

Quelle est la complexité de l'extraction du minimum ?

### 5.3.6 Fusion

Etant donnés deux tas binomiaux, il est possible de les fusionner en utilisant comme sous-programme la fusion de deux arbres.

**Exercice 16 - Fusion**

Décrire une méthode récursive permettant de fusionner deux tas binomiaux. Quelle en est la complexité ?

**Exercice 17 - Autres opérations**

Décrire de quelle façon ramener à une fusion l'insertion d'un nouvel élément. Que se passe-t-il si l'on supprime la racine d'un arbre binomial ? En déduire un moyen de ramener la suppression d'un élément à une fusion. Déterminer les complexités de ces deux opérations.

# Chapitre 6

# Hachage

## 6.1 Principe

Une table de hachage est une structure de données indexée par des valeurs $V = \{1, \ldots, m\}$, comme un tableau. On dit alors que cette table est de taille $m$. Une table de hachage contient des éléments posédant chacun une clé, on note $C = \{1, \ldots, n\}$ l'ensemble des clés. Il existe en règle générale beaucoup plus de clés que d'indices dans une table. On affecte à chaque élément, donc à chaque clé, un (ou plusieurs) indice dans la table. Pour ce faire, on dispose d'une **fonction de hachage** $h : C \longrightarrow V$, qui à toute clé $c$ associe un indice $h(c)$ permettant de placer (rechercher, ou supprimer) un élément dans la table de hachage.

## 6.2 Collisions

Comme $|C| > |V|$, $h$ ne peut être injective. Par conséquent, il est possible que des éléments se téléscopent, c'est-à-dire que deux clés distinctes aient la même image par $h$. Deux techniques existent pour gérer les collisions :

1. Le chaînage
2. L'adressage ouvert

### 6.2.1 Gestion des collisions par chaînage

Pour résoudre les collisions par chaînage, on place dans chaque emplacement de la table non pas les éléments eux-mêmes, mais un pointeur vers une liste chaînée contenant ces éléments. Si un emplacement est déjà ocupé par un ensemble d'éléments $T$ au moment de l'insertion de $e$, alors on place $e$ à la fin de la liste chaînée formée par les éléments de $e$.

### 6.2.2 Gestion des collisions par adressage ouvert

L'adressage ouvert consiste à étendre la fonction de hachage en un ensemble de fonctions $\{h_i | i \in V\}$, lors de l'insertion d'un élément $e$, on applique d'abord $h_1$ à la clé de $e$, si l'emplacemnet est disponible, alors on y place $e$, sinon on lui cherche un nouvel emplacement avec la fonction $h_2$, etc.

Nous ne travaillerons dorévavant que sur des tables dans lesquelles les collisions sont gérées par chaînage.

#### Exercice 1 - Résolution par chaînage

Soit $h$ la fonction de hachage définie par $h(k) = kmod8$ et soit une table de hachage de taille $m = 8$ dans laquelle les collisions sont résolues par chaînage. Donner l'état de la table après l'insertion des clés 5, 28, 19, 15, 20, 33, 12, 17 et 10.

**Exercice 2 - Temps moyen de recherche d'un noeud**

On considère une fonction de hachage $h$ qui répartit uniformément les clés dans une table de hachage de taille $m$ dans laquelle les collisions sont résolues par chaînage.

1. Si $n$ clés sont présentes dans la table, quelle est la longueur moyenne d'une liste.
2. Quelle est la durée moyenne d'une recherche infructueuse dans la table ?
3. On part du principe que si $r(x)$ est le rang d'insertion de la clé $x$ dans la table, les $n$ valeurs que peut prendre $r(x)$ sont équiprobables. Déterminer la durée moyenne d'une recherche de $x$.
4. Quelle est la durée moyenne de recherche d'une clé présente dans la table ?
5. Quelle est la durée moyenne moyenne de recherche d'une clé dans une table de hachage de taille $m$ contenant $\mathcal{O}(m)$ clés ?

# Chapitre 7

# AVL

## 7.1 Arbres binaires de recherche

**Définition 7.1.1** *Soit $A$ un arbre binaire dont chaque noeud est muni d'une clé. $A$ est un arbre binaire de recherche si pour tout noeud $x$,*

- $\forall y \in A_g(x)$, $cle(y) < cle(x)$
- $\forall z \in A_d(x)$, $cle(x) < cle(z)$

Pour davantage d'informations, voir [2].

**Exercice 1 - Définition**

Construire tous les ABR contenant les valeurs $\{1, 2, 3\}$

**Exercice 2 - Notation ensembliste**

1. Ecrire un algorithme plaçant dans une variable $v$ l'arbre $((\emptyset, 3, \emptyset), 1, (\emptyset, 2, \emptyset))$
2. Dessiner le graphe obtenu
3. Est-ce un ABR bien formé ?

**Exercice 3 - Insertions**

1. Dessiner l'arbre $A(v)$ défini comme suit :
   - $a = (\emptyset, 4, \emptyset)$
   - $b = ((\emptyset, 11, \emptyset), 7, \emptyset)$
   - $c = (\emptyset, 3, (\emptyset, 9, \emptyset))$
   - $d = ((\emptyset, 12, \emptyset), 6, (\emptyset, 2, \emptyset))$
   - $v = ((a, 5, b), 1, (c, 10, d))$
2. Est-ce un ABR bien formé ?
3. Changer l'ordre des noeuds de sorte que $A(v)$ devienne un ABR.
4. Insérez le noeud de clé 8 dans cet arbre

**Exercice 4 - Suppression**

On notera

- $succ_A(x)$ le successeur de $x$ dans $A$, c'est-à-dire l'élément de $A(x)$ de plus petite clé parmi les éléments de clé supérieure à celle de $x$.
- $premier(A(x))$ le plus petit élément de $A(x)$

1. Prouvez que si un noeud $x$ d'un arbre $A$ a deux fils, alors $succ_A(x)$ n'a pas de fils gauche.
2. Montrez que $y = succ_A(x)$ si et seulement si $y = premier(A_d(x)$ ou $x = dernier(A_g(y))$.
3. Supprimez la racine de l'arbre de l'exercice précédent

**Exercice 5 - Fonctions récursives**

1. Ecrire une fonction insérant une clé $x$ dans un arbre $A$ et retournant l'arbre obtenu.
2. Ecrire une fonction retournant la clé minimale de l'arbre $A$.
3. Ecrire une fonction prenant un arbre $A$ en paramètre et retournant le couple ($A$ privé de sa clé minimale, la clé minimale de $A$).
4. Ecrire une fonction retournant vrai si et seulement si l'arbre $A$ passé en paramètre est un ABR bien formé.
5. Ecrire une fonction prenant en paramètres un arbre $A$ est une valeur $c$, et retournant l'arbre $A$ dans lequel aura été supprimé le noeud de clé $c$.

**Exercice 6 - Application au tri**

Ecrire une fonction $triAbr(l)$ prenant en paramètre une liste $l$ contenant des éléments d'un ensemble totalement ordonné et retournant un liste contenant les éléments de la liste $l$ triés par ordre croissant. Vous définirez pour ce faire toutes les fonctions auxiliaires nécessaires pour rendre l'algorithme lisible.

## 7.2 Complexité dans le pire de cas

La complexité dans le pire des cas est peu satisfaisante. Insérons dans un arbre des éléments par ordre croissant, et il seront disposés en liste. Chaque opération se fera de la sorte en un temps $\mathcal{O}(n)$. Ces performances sont les mêmes qu'avec des listes chaînées. On en conclut que dans le pire des cas, les ABR ne sont pas plus performants que des vulgaires listes chaînées. Nous allons nous intéresser, dans la suite de ce cours, à des arbres binaires de recherche quelque peu particuliers, dans lesquels toute opération se fera en temps $\mathcal{O}(log_2(n))$.

**Exercice 7 - complexité moyenne d'une recherche**

Soit un ensemble $E = \{1, \ldots, n\}$ de $n$ clés. Nous voulons déterminer le nombre moyen $a_n$ de noeuds visités lors de la recherche d'une de ces clés dans un ABR contenant les clés de $E$. On suppose que les ordres d'insertion sont équiprobables. Par ailleurs, on considère que la probabilité de rechercher la clé $i \in E$ est $\frac{1}{n}$.

1. Montrer que la probabilité que la racine ait la clé $i$ est $\frac{1}{n}$.
2. Notons $a_n^i$ le nombre moyen de comparaisons nécessaires pour trouver une clé dans un arbre à $n$ clés dont la racine porte la clé $i$. Exprimer $a_n$ en fonction de $\{a_n^i | i \in E\}$.
3. On pose $a_0 = 0$. Combien valent $a_1^1$ et $a_1$ ?
4. Montrer que $\forall i \in E, a_n^i = (a_{i-1} + 1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i} + 1)\frac{n-i}{n}$
5. En déduire que $a_n = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i a_i$
6. En déduire que $a_n = \frac{1}{n^2}((n^2-1)a_{n-1} + 2n - 1)$
7. Utiliser le fait que $log_2(n-1) = \frac{1}{ln2}\int_1^{n-1} \frac{dx}{x}$ et que $\frac{1}{n} \leq \int_{n-1}^{n} \frac{dx}{x}$ pour prouver par récurrence qu'il existe $\alpha$ tel que $a_n \leq \alpha log_2(n)$
8. Quelle est la complexité moyenne de la recherche d'une clé de $E$ dans un ABR ?

## 7.3 Rotations

Nous allons étudier une opération sur les ABR s'appelant la rotation (je vous conseille de faire des dessins si vous tenez à vous représenter les choses concrètement). Il existe deux rotations simples (la rotation droite et la rotation gauche), et deux rotations doubles (la rotation droite-gauche et la rotation gauche-droite). La rotation droite est définie de la sorte :

$$rotationDroite(((B, y, C), x, D)) = (B, y, (C, x, D))$$

De façon symétrique, on définit la rotation gauche :

$$rotationGauche((B, x, (C, y, D)) = ((B, x, C), y, D)$$

La rotation gauche-droite se définit à l'aide des deux précédentes :

$$rotationGaucheDroite((G, x, D)) = rotationDroite(rotationGauche(G), x, D)$$

Et on définit pour finir la rotation droite-gauche de façon symétrique

$$rotationDroiteGauche((G, x, D)) = rotationGauche(G, x, rotationDroite(D))$$

## 7.4 AVLs

Si pour un noeud $v$ d'un ABR, les hauteurs des sous-arbres droit et gauche de $v$ diffèrent de au plus 1, on dit que ce noeud est équilibré. Un AVL est un ABR dans lequel tous les noeuds sont équilibrés.

### 7.4.1 Rééquilibrage

Les rotations vont servir à maintenir l'équilibre de chaque noeud. Nous allons nous intéresser à l'équilibre d'un arbre de racine $v$ dont les deux sous-arbres ($G$ et $D$) sont équilibrés, et ont des hauteurs qui diffèrent de 2. Supposons, sans perte de généralité, que $h(D) = n$, et que $h(G) = n + 2$. Soit $G = (B, y, D)$, alors deux cas se présentent :

- si $h(B) = n + 1$, alors une rotation droite de $v$ équilibre l'arbre.
- si $h(B) \neq n + 1$, alors une rotation gauche-droite de $v$ équilibre l'arbre.

### 7.4.2 Insertion

Pour insérer un noeud, procédez comme avec des ABR pour l'ajouter en tant que feuille. Ensuite, contrôlez l'équilibre de chaque noeud et faites des rotations si nécessaire sur le chemin allant de la feuille nouvellement insérée à la racine.

### 7.4.3 Suppression

De la même façon qu'avec des ABR, on supprime un noeud en le remplaçant par le noeud du plus grande clé du sous-arbre gauche, ou celui de plus petite clé du sous-arbre droit. La suppression du minimum ou du maximum a les mêmes conséquences que pour l'insertion, il est nécessaire de contrôler l'équilibre de chaque noeud sur le chemin menant du père de la feuille supprimée à la racine.

### 7.4.4 Compléments

Pour plus de détails, je vous conseille de vous reporter à [3]. L'applet [4] illustre graphiquement le fonctionnement d'un AVL.

#### Exercices

#### Exercice 8 - Implémentation en C

Ecrivez les fonctions du fichier `avl.c` correspondant au fichier `avl.h` ci-dessous :

```
#ifndef AVL_H

#include"linkedList.h"

#define AVL_H

typedef struct
{
```

```
  /*
    Pointeur vers une fonction permettant de
    recuperer la cle de chaque donnee.
   */
  int (*getKey)(void*);

  /*
    Pointeur vers une fonction permettant de
    de detruire chaque donnee.
  */
  void (*freeData)(void*);

  /*
   Pointeur vers la racine de l'arbre.
  */
  void* root;
}avl ;

/*-------------------------------------------------------*/

/*
  Retourne un AVL vide.
*/

avl* avlCreate(int (*getKey)(void*), void (*freeData)(void*));

/*
  Modifie la fonction de destruction des donnees.
*/

void avlSetFreeFunction(avl* a, void (*freeData)(void*));

/*
  Affiche les cles de l'AVL a dans l'ordre croissant, O(n).
*/

void avlPrintKeys(avl* a);

/*
  Insere la donnee v dans l'AVL a, O(log n).
*/

void avlInsert(avl* a, void* v);

/*
  Retourne la donnee de cle x si elle se trouve dans l'AVL a,
  NULL sinon, O(log n).
*/

void* avlFind(avl* a, int x);

/*
  Supprime le noeud de cle k de l'AVL a, applique la fonction
  de destruction a la donnee de cle k, O(log n).
*/

void avlRemove(avl* a, int k);

/*
  Detruit l'AVL a, applique la fonction de destruction a
  toutes les donnees du sous-arbre, O(n).
*/

void avlDestroy(avl* a);

/*
  Retourne une liste chainee contenant toutes les donnees
  de l'AVL a disposees dans l'ordre croissant, O(n).
 */

linkedList* avlToList(avl* a);

#endif
```

Et voici `avl.c`.

```
#include<stdio.h>
#include<malloc.h>
#include"avl.h"

/*-------------------------------------------------------*/
```

```
/*
  Noeud de l'ABR.
 */

typedef struct nd
{
  /*
    key est la cle du noeud courant.
   */
  int key;

  /*
    pointeur vers l'element de cle key
   */
  void* data;

  /*
    hauteur du sous-arbre :
    0 si ce noeud est une feuille.
   */
  int height;

  /*
    Pointeur vers le sous-arbre droit.
   */
  struct nd * left;

  /*
    Pointeur vers le sous-arbre gauche.
   */
  struct nd * right;
}node;

/*--------------------------------------------------------*/

/*
  Retourne la hauteur de l'arbre de racine l,
  -1 si l est vide.
 */

int getHeight(node* l)
{
}

/*--------------------------------------------------------*/

/*
  Retourne la plus grande des deux valeurs i et j.
 */

int max(int i, int j)
{
}

/*--------------------------------------------------------*/

/*
  Met a jour la hauteur de la racine l en fonction des
  hauteurs des racines des deux sous-arbres.
 */

void setHeight(node* l)
{
}

/*--------------------------------------------------------*/

/*
  Cree un noeud contenant la donnee data,
  ayant pour sous-arbre gauche l et
  pour sous-arbre droit r.
*/

node* nodeCreate(int (*getKey)(void*), node* l, void* data, node* r)
{
}

/*--------------------------------------------------------*/

/*
  Retourne un avl vide
```

```
*/

avl* avlCreate(int (*getKey)(void*), void (*freeData)(void*))
{
}

/*-------------------------------------------------------*/

/*
  Fait de freeData la fonction de destruction des donnees.
*/

void avlSetFreeFunction(avl* a, void (*freeData)(void*))
{
}

/*-------------------------------------------------------*/

/*
  Affiche toutes les cles du sous-arbre de racine n dans
  l'ordre croissant.
*/

void nodePrintKeys(node* n)
{
}

/*-------------------------------------------------------*/

/*
  Affiche pour tous les noeuds du sous-arbre de racine n
  la difference entre les hauteurs du sous-arbre droit et
  gauche.
*/

void nodePrintLevels(node* n)
{
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation droite de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateRight(node* x)
{
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation gauche de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateLeft(node* x)
{
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation gauche-droite de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateLeftRight(node* x)
{
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation droite-gauche de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateRightLeft(node* x)
{
```

```
}

/*-------------------------------------------------------------*/

/*
  Reequilibre l'arbre de racine x, retourne la racine de
  l'arbre apres reequilibrage. On part du principe
  que les hauteurs des sous-arbres droit et gauche different
  de au plus 1.
*/

node* balance(node* x)
{
}

/*-------------------------------------------------------------*/

/*
  Insere la donnee v dans l'arbre de racine n
  et reequilibre l'arbre, retourne la racine de
  l'arbre apres insertion et reequilibrage.
 */

node* nodeInsert(int (*key)(void*), node* n, void* v)
{
}

/*-------------------------------------------------------------*/

/*
  Insere la donnee v dans l'arbre a, O(log n).
*/

void avlInsert(avl* a, void* v)
{
}

/*-------------------------------------------------------------*/

/*
  Affiche les cles de l'AVL a dans l'ordre croissant, O(n)
*/

void avlPrintKeys(avl* a)
{
}

/*-------------------------------------------------------------*/

/*
  Affiche pour chaque noeud la difference entre les hauteurs
  des sous-arbres droit et gauche de l'AVL.
*/

void avlPrintLevels(avl* a)
{
}

/*-------------------------------------------------------------*/

/*
  Retourne la donnee de cle k si elle se trouve dans le sous-arbre
  de racine n, NULL sinon.
*/

void* findNode(node* n, int k)
{
}

/*-------------------------------------------------------------*/

/*
  Retourne la donnee de cle x si elle se trouve dans l'AVL a,
  NULL sinon, O(log n)
*/

void* avlFind(avl* a, int x)
{
}

/*-------------------------------------------------------------*/
```

```c
/*
  Fait pointer *max vers le noeud de cle maximale dans
  le sous-arbre de racine n, detache le noeud *max de l'arbre
  et retourne la racine de l'arbre apres suppression.
*/

node* removeMax(node* n, node** max)
{
}

/*-------------------------------------------------------------*/

/*
  Fait pointer *min vers le noeud de cle minimale dans
  le sous-arbre de racine n, detache le noeud *min de l'arbre
  et retourne la racine de l'arbre apres suppression.
*/

node* removeMin(node* n, node** min)
{
}

/*-------------------------------------------------------------*/

/*
  Supprime le noeud de cle k du sous-arbre de racine n,
  applique la fonction de destruction a la donnee de ce noeud,
  et retourne la racine de l'arbre apres la suppression.
*/

node* removeNode(node* n, int k, void (*freeData)(void*))
{
}

/*-------------------------------------------------------------*/

/*
  Supprime le noeud de cle k de l'AVL a, applique la fonction
  de destruction a la donnee de cle k, O(log n).
*/

void avlRemove(avl* a, int k)
{
}

/*-------------------------------------------------------------*/

/*
  Place dans la liste chainee l toutes les donnees
  du sous-arbre de racine n dans l'ordre croissant.
 */

void nodeToList(node* n, linkedList* l)
{
}

/*-------------------------------------------------------------*/

/*
  Retourne une liste chainee contenant toutes les donnees
  de l'AVL a disposees dans l'ordre croissant, O(n).
 */

linkedList* avlToList(avl* a)
{
}

/*-------------------------------------------------------------*/

/*
  Detruit tous les noeuds du sous-arbre de racine n,
  applique la fonction de destruction fr a toutes les
  donnees du sous-arbre.
*/

void nodeDestroy(node* n, void (*fr)(void*))
{
}

/*-------------------------------------------------------------*/
```

```c
/*
  Detruit l'AVL a, applique la fonction de destruction a
  toutes les donnees du sous-arbre, O(n).
*/

void avlDestroy(avl* a)
{
}

/*--------------------------------------------------------*/
/*--------------------------------------------------------*/

/*
Pour tester les algorithmes
*/

void destroyInt(void* i)
{
  free((int*)i);
}

int getIntKey(void* i)
{
  return *((int*)i);
}

void doNothing()
{

}

int main()
{
  int i;
  int* d;
  linkedList* l;
  link* m;
  avl* a = avlCreate(getIntKey, destroyInt);
  for (i = 0 ; i < 40 ; i++)
    {
      d = (int*)malloc(sizeof(int));
      *d = i;
      avlInsert(a, d);
    }
  avlPrintKeys(a);
  avlPrintLevels(a);
  for(i = 0 ; i < 40 ; i+=5)
    avlRemove(a, i);
  avlPrintKeys(a);
  avlPrintLevels(a);
  l = avlToList(a);
  for(m = linkedListGetFirst(l); m != NULL ; m = m->next)
    printf("%d -> ", *(int*)(m->data));
  printf("\n");
  linkedListDestroy(l, doNothing);
  avlSetFreeFunction(a, destroyInt);
  avlDestroy(a);
  return 0;
}
```

# Chapitre 8

# Modélisation

## 8.1 Définition

Modéliser est une des compétences les plus importantes en algorithmique. A partir d'un problème concret (industriel, logistique, économique, etc.), on effectue une modélisation en formalisant le problème avec une terminologie mathématique, et en reformulant la question de façon précise. Meilleure est la modélisation, plus il est simple de résoudre le problème.

## 8.2 Exemple

Voici un exemple de problème : comment placer 8 reines sur un échiquer de sorte qu'aucune ne puisse s'emparer d'une autre ? Si vous souhaitez réaliser un programme qui résolve ce problème, vous allez d'abord devoir le modéliser mathématiquement. Dans la plupart des problèmes, on a

- **des données** : les emplacement des reines
- **des contraintes** : toutes les reines sont placées, il n'existe pas de couple de reines sur la même ligne, la même colonne, ou la même diagonale
- **une question** : est-il possible de placer ces huit reines sans violer de contrainte ?

Il faut donc commencer par réfléchir à une façon de modéliser les emplacement des reines.

### 8.2.1 Données

#### Première modélisation

On pourrait, naïvement prendre une matrice $M$ de dimensions $8 * 8$, et placer 0 dans une case s'il ne s'y trouve pas de reine, 1 s'il s'en trouve une.

#### Première modélisation

Ou bien considérer huit couples de données qui contiendraient les coordonnées de la case sur laquelle serait placée chaque reine. Ainsi, la $i$-ème reine occupe la case $(l_i, c_i)$

#### Première modélisation

Ou plus simplement, puisqu'il n'y a qu'une reine par ligne, un vecteur de 8 valeurs nous donnerait la colonne sur laquelle serait placée chaque reine. $c_i$ est donc la colonne sur laquelle on a placé la reine de la ligne $i$.

#### Bref

Considérons comme critère la place mémoire occupée, il va de soi que ces solutions sont de qualité croissante. Nous allons voir qu'elles le sont aussi quand il s'agit de poser les contraintes.

### 8.2.2 Contraintes

La façon d'exprimer les contraintes dépend de la façon dont ont été représentées les données.

**Première modélisation**

- toutes les reines sont placées : $\sum_{i=1}^{8}\sum_{j=1}^{8} m_{ij} = 8$
- une reine par ligne : $\forall i \in \{1,\dots,8\}, \sum_{i=1}^{8} m_{ij} = 1$
- une reine par colonne : $\forall j \in \{1,\dots,8\}, \sum_{j=1}^{8} m_{ij} = 1$
- pas plus d'une reine par diagonale : $\forall p \in \{2,\dots,16\}, \sum_{i+j=p} m_{ij} \leq 1$ et $\forall p \in \{-7,\dots,7\}, \sum_{i-j=p} m_{ij} \leq 1$.

compliqué, non ?

**Deuxième modélisation**

- toutes les reines sont placées : rien à faire
- une reine par ligne : $\forall i \in \{1,\dots,8\}, \forall j \in \{1,\dots,8\}$ tel que $i \neq j, l_i \neq l_j$
- une reine par colonne : $\forall i \in \{1,\dots,8\}, \forall j \in \{1,\dots,8\}$ tel que $i \neq j, c_i \neq c_j$
- pas plus d'une reine par diagonale : $\forall i \in \{1,\dots,8\}, \forall j \in \{1,\dots,8\}$ tel que $i \neq j, |c_i - c_j| \neq |l_i - l_j|$

C'est plus simple, non ?

**Troisième modélisation**

- toutes les reines sont placées : rien à faire
- une reine par ligne : rien à faire
- une reine par colonne : $\forall i \in \{1,\dots,8\}, \forall j \in \{1,\dots,8\}$ tel que $i \neq j, c_i \neq c_j$
- pas plus d'une reine par diagonale : $\forall i \in \{1,\dots,8\}, \forall j \in \{1,\dots,8\}$ tel que $u \neq j, |i - j| \neq |c_i - c_j|$

Encore plus simple.

**Bref**

Vous remarquez que ces modèles sont aussi de qualité croissante si on prend comme critère la simplicité de l'expression des contraintes.

### 8.2.3 Question

La question est à chaque fois la même : quelles valeurs donner aux variables pour ne violer aucune contrainte ? Mais le nombre de jeux de valeurs à donner aux variables n'est pas le même à chaque fois.

1. $2^{64}$ possibilités, ce qui est de loin supérieur au nombre d'atomes dans l'univers...
2. $(8^2)^8$ possibilités, c'est déjà mieux, ça fait $((2^3)^2)^8 = 2^{48}$ combinaisons possibles.
3. $8^8$ possibilités, à savoir $(2^3)^8 = 2^{24}$ combinaisons, ce qui représente seulement quelques millions de possibilités...

Parmi les modélisations proposées, la dernière est donc la meilleure sur tous les plans. Seul inconvénient pour le moment, on ne voit pas comment résoudre ce problème sans énumérer toutes les possibilités... Nous répondrons à cette question dans les cours suivants.

## 8.3 Sensibilisation à la complexité

### 8.3.1 Le tour de la table

Vous vous attablez deux fois par jour avec votre petite famille de seulement 8 personnes pour manger. Il vous vient l'idée de changer de place à chaque repas de sorte que l'on ait jamais deux fois la même configuration (i.e. toutes les

personnes à la même place). Combien de temps vous faudra-t-il pour vous attabler suivant toutes les configurations possibles ?

### 8.3.2 De la terre à la Lune

Vous disposez d'une feuille de papier d'un dixième de millimètre d'épaisseur. Vous décidez de la plier plusieurs fois sur elle-même (tant que c'est possible). A votre avis, est-il possible de faire 10 pliages, où votre feuille aura-t-elle la même épaisseur que le Petit Robert ? Pourrez-vous effectuer 30 pliages, ou l'épaisseur de votre feuille sera-t-elle comparable à la taille d'une terrain de football ? Et en supposant que vous soyez capable de la plier 50 fois sur elle-même, à combien de fois la distance de la terre à la Lune l'épaisseur de cette feuille sera-t-elle égale ?

### 8.3.3 En bref

Lorsque vous cherchez des solutions à des problèmes combinatoires, bannissez d'entrée de jeu toute méthode basée sur une exploration exhaustive de l'ensemble des solutions.

## 8.4 Terminologie

### 8.4.1 Difficulté

Nous allons passer en revue dans la section suivante quelques problèmes classiques d'optimisation combinatoire. Nous les classerons en deux catégories

- **Les problèmes faciles** : c'est-à-dire ceux qui se résolvent en temps polynomial, autrement dit sans avoir à explorer l'intégralité des solutions.
- **Les problèmes difficiles** : c'est-à-dire ceux qu'on ne peut résoudre qu'en explorant toutes les solutions. On dit que ces problèmes sont **NP-Complets**.

La définition ci-dessus est très shématique, tenez-vous en à celle-ci pour le moment, un cours sera ultérieurement consacré à l'etude des problèmes NP-Complets.

### 8.4.2 Rappels de théorie des graphes

La plupart des problèmes exposés ci-après sont des problèmes de théorie des graphes. Un graphe $G$ est un couple $(S, A)$ où $S$ est l'ensemble des sommets (ou noeuds) et $A$ l'ensemble des arêtes (ou arcs). Nous ne verrons que des exemples dans lesquels l'ensemble $S$ est fini, sachant qu'il peut contenir n'importe quoi (des entiers, des réels, des couples de matrices, ds sous-ensembles de $\mathbb{R}$, des graphes, etc.). L'ensemble $A$ par contre ne peut contenir que des couples (si le graphe est orienté, des paires si le graphe n'es pas orienté) d'éléments de $S$. Par exemple, $G = (S, A)$ avec

$$S = \{(1,2), (2,3), (1,3), (4,5), (5,4)\}$$

et

$$A = \{((1,2),(2,3)), ((1,2),(1,3)), ((4,5),(5,4)), ((5,4),(4,5)), ((1,3),(1,3)), ((2,3),(1,2))\}$$

est un graphe (orienté en l'ocurrence). Si un graphe est orienté, on ne dit plus sommets mais noeuds, et on ne dit plus arêtes mais arcs.

Les graphes sont quelquefois valués, c'est-à-dire qu'il existe une application de $A$ dans un sous-ensemble de $\mathbb{R}$. Ils sont quelquefois pondérés, c'est-à-dire qu'il existe une application de $S$ dans un sous-ensemble de $\mathbb{R}$.

### 8.4.3 Définition d'un problème

La plupart des problèmes suivants sont définis en deux temps :

- des **données**
- une **question**

Les questions permettent de classer les problèmes en plusieurs catégories :

- les problèmes de **décision** : questions de la forme "existe-t-il...", la réponse est oui ou non.
- les problèmes de **satisfaction de contraintes** : questions de la forme "trouver une solution vérifiant...". La réponse doit juste satisfaire des contraintes.

– les problèmes d'**optimisation** : questions sous la forme "trouver ... minimal (ou maximal) tel que...". La réponse doit satisfaire des contraintes et être optimale.

### 8.4.4 Problèmes d'optimisation

Dans un problème d'optimisation, la valeur que l'on cherche à maximiser, ou à minimiser, s'appelle la **fonction objectif**. Une solution d'un problème d'optimisation est dite
– **réalisable** si elle ne viole aucune contrainte.
– **optimale** si elle est la meilleure des solutions réalisables.

Notez qu'il peut y avoir plusieurs solutions optimales. Tout comme il peut ne pas y avoir de solution réalisable, dans ce cas le problème d'optimisation n'a pas de solution.

## 8.5 Problèmethèque

### 8.5.1 Arbre couvrant de poids minimal

Soit $G = (S, A)$ un graphe non orienté muni d'une valuation des arêtes $c$, $T = (S_T, S_A)$ est un arbre couvrant $G$ si
– $T$ est un arbre
– $S_T = S$
– $T$ est connexe

Le poids de $T$ est

$$\sum_{a \in A_T} c(a)$$

Le problème de l'arbre couvrant de poids minimal est le suivant :
– **données** : un graphe $G$, une valuation $c$
– **question** : trouver l'arbre $T$ de plus petit poids parmi les arbres couvrant $G$.

Ce problème (d'optimisation) est facile, les deux algorithmes les plus connus sont ceux de Primm et de Kruskall.

### 8.5.2 Ensemble stable maximal

Soit $G = (S, A)$ un graphe non orienté muni d'une pondération $p$ un sous-ensemble $K$ de $S$ est un ensemble stable si

$$\forall a = (e, e') \in A, e \notin K \ ou \ e' \notin K$$

Le poids de cet ensemble stable est

$$\sum_{s \in K} p(s)$$

Le problème de l'ensemble stable maximal est défini comme suit :
– **données** : un graphe $G$, une pondération $p$
– **question** : trouver le sous-ensemble $K$ de poids maximal parmi les ensembles stables de $G$.

Ce problème est NP-Complet dans le cas général, polynomial si le graphe est un arbre ou est biparti

### 8.5.3 Circuit eulérien

Etant donné un graphe $G$ orienté ou non, un circuit eulérien dans $G$ est un circuit passant par une et une seule fois par chaque **arête** de $G$. Le problème est le suivant :
– **données** : un graphe $G$
– **question** : trouver un circuit eulérien dans $G$.

Ce problème est polynomial, il se résoud avec un algorithme inspiré par les travaux d'Euler sur les graphes.

### 8.5.4 Circuit hamiltonien

Soit $G$ un graphe (orienté ou non), un circuit hamiltonien dans $G$ est un circuit passant par une et une seule fois par chaque **sommet** de $G$. Le problème du circuit hamiltonien est le suivant :

- **données** : un graphe $G$
- **question** : trouver un circuit hamiltonien dans $G$.

Ce problème est NP-Complet.

### 8.5.5 Voyageur de commerce

Soit $G$ un graphe (orienté ou non) valué par une fonction $c$, le poids d'une chemin (ou d'un circuit) $K$ la somme, pour toute arête $a$ de $K$, des $c(a)$.

- **données** : un graphe $G$
- **question** : trouver un circuit hamiltonien de poids minimal dans $G$.

Ce problème est NP-Complet.

### 8.5.6 SAT

Soit une expression booléenne à $n$ variables sous forme clausale (conjonction de disjonctions), par exemple,

$$(x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_3) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4)$$

Les clauses (contenant des disjonctions) sont notées

$$\{C_1, c_2, \ldots, c_k\}$$

Toute clause est formée de $k_i$ littéraux

$$C_i = l_1^i, l_2^i, \ldots, l_{k_i}^i$$

chaque littéral $l$ est soit une variable ($l = x_j$), soit la négation d'une variable ($l = \not{x}_j$). Une affectation $\mathcal{A}$ des variables est une application de $\{x_1, \ldots, x_n\}$ dans $\{true, false\}$.

- Si $\mathcal{A}(x_j) = true$, alors $l$ est satisfait si $l = x_j$, non satisfait sinon
- si $\mathcal{A}(x_j) = false$, alors $l_j$ est satisfait si $l = \neg x_j$, non satisfait si $l = x_j$

Une clause est satisfaite si au moins un des littéraux qui la forment est satisfait. Une expression sous forme clausale est satisfaite si toutes les clauses sont satisfaites. Le problème est le suivant :

- **données** : $n$ variables, $k$ clauses de chacune $k_i$ littéraux, chacun correspondant à une variable ou à la négation d'une variable.
- **question** : trouver une affectation qui satisfasse toutes les clauses (donc au moins un littéral par clause)

Ce problème (de satisfaction de contraintes) est NP-Complet dans le cas général, et facile si les clauses contiennent au plus 2 littéraux.. Remarquons que si la question avait été "existe-t-il une affectation qui satisfasse toutes les clauses ?", ce problème serait un problème de décision.

### 8.5.7 Couverture

Soit $G = (S, A)$ un graphe non orienté muni d'une pondération $p$ un sous-ensemble $K$ de $S$ est une couverture si

$$\forall a = (e, e') \in A, e \in K \; ou \; e' \in K$$

Le poids de cette couverture est

$$\sum_{s \in K} p(s)$$

Le problème de la couverture minimale est défini comme suit :

- **données** : un graphe $G$, une pondération $p$
- **question** : trouver le sous-ensemble $K$ de poids minimal parmi les couvertures de $G$.

Ce problème est NP-Complet dans le cas général, polynomial si le graphe est un arbre ou est biparti.

### 8.5.8 Coloration

Soit $G = (S, A)$ un graphe non orienté, une $k$-coloration $c$ de $g$ est une application de $S$ dans un ensemble $E$ tel que $|E| = k$ vérifiant

$$\forall (e, e') \in A, c(e) \neq c(e')$$

Le problème de la $k$-coloration est défini comme suit :
- **données** : un graphe $G$
- **question** : trouver une $k$-coloration de $G$.

Ce problème est NP-Complet.
Le nombre chromatique $\chi_G$ d'un graphe est la plus petite coloration de ce graphe. Autrement dit,

$$\chi_G = \min\{k | G \; est \; k - colorable\}$$

- **données** : un graphe $G$
- **question** : trouver le nombre chromatique de $G$.

Ce problème est aussi NP-Complet.

### 8.5.9 Plus court chemin

Soit $g$ un graphe orienté (ou non) muni d'une valuation de arêtes $c$. Un chemin $C$ de longueur $p$ dans $G$ est un $p$-uplet $(c_1, \ldots, c_p)$ d'éléments de $A$ tel que si $c_i = (s_i, t_i)$, alors pour tout $i \in \{1, \ldots, p-1\}$, $t_i = s_{i+1}$. $C$ est un chemin de $u \in S$ à $v \in S$ si $s_1 = u$ et $t_p = v$.

Le problème du plus court chemin dans un graphe valué se définit comme suit :
- **données** : un graphe $G$ valué, deux sommets $u$ et $v$
- **question** : trouver le plus court chemin de $u$ à $v$ dans $G$.

Ce problème est polynomial, Les deux algorithmes les plus connus sont
- **Dijkstra** : très rapide, $\mathcal{O}(|S| + |A|)$ dans le pire des cas, mais ne fonctionne qu'avec les valuations positives, ne nécessite pas le parcours de tout le graphe.
- **Bellman** : un peu moins rapide, il présente l'avantage de fonctionner avec des valuations négatives, sauf si le graphe contient un circuit absorbant (i.e. négatif).

### 8.5.10 Sac à dos

Etant donné un ensemble $A = \{a_1, \ldots, a_n\}$, des poids positifs$\{p_i | i \in \{1, \ldots, n\}\}$, des valeurs positives $\{v_i | i \in \{1, \ldots, n\}\}$, et une constante positive $K$. Une solution réalisable est un sous-ensemble $I$ de $\{1, \ldots, n\}$ tel que $\sum_{i \in I} p_i \leq K$, la valeur de la fonction objectif est $\sum_{i \in I} v_i$. Le problème du sac à dos est le suivant :

- **données** : un ensemble $A$ muni de deux valuations $\{p_i | i \in \{1, \ldots, n\}\}$ et $\{v_i | i \in \{1, \ldots, n\}\}$, et une constante $K > 0$
- **question** : trouver le sous ensemble $B$ de $A$ maximisant $\sum_{i \in I} v_i$ sous la contrainte $\sum_{i \in I} p_i \leq K$.

Ce problème est NP-Complet.

## 8.6 Exercices

**Exercice 1 - Digicode**

Vous souhaitez craker un digicode, vous devez pour cela trouver un code de $k$ chiffres, le digicode en comportant $n$.

1. Combien de chiffres seront, dans le pire des cas, saisies sur le clavier si vous décidez d'énumérer tous les codes les uns à la suite des autres, et suivant un ordre lexicographique ?
2. Si vous décidez d'entremêler les codes les uns dans les autres, donnez un minorant de la meilleure méthode envisageable.

3. Modélisez ce problème comme un problème de recherche d'un circuit hamiltonnien dans un graphe orienté. La résolution est-elle facile ?
4. Modélisez ce problème comme un problème de recherche d'un circuit eulérien dans un graphe, le problème est-il facile ?

**Exercice 2 - Rattrapages**

Un ensemble $E = \{e_1, \ldots, e_n\}$ doivent passer des sessions de rattrapages dans les matières $M = \{m_1, \ldots, m_k\}$. Vous disposez d'un ensemble de couples $C = (c_1, \ldots, c_j)$ tel $c_i = (e, m)$ signifie que l'étudiant $e$ doit repasser la matière $m$. Nous souhaitons organiser les rattrapages en affectant chaque épreuve à un jour. Plusieurs épreuves puissent se dérouler le même jour tant qu'un même candidat n'a pas deux ratrappages à effectuer le même jour. Nous souhaitons minimiser le nombre de jours que dureront les rattrapages.

1. Comment modéliser le problème de la façon la plus simple possible ?
2. Est-il polynomial ?

**Exercice 3 - Gardiens de prison**

Une prison est contituée de couloirs qui s'intersectent en de multiples points. Le directeur souhaiterait, avec un minimum de gardiens, surveiller les couloirs. il faudrait pour cela, placer les gardiens aux intersections des couloirs de sorte que chaque couloir soit visible par au moins un gardien. Comment modéliser ce problème en théorie des graphes ?

**Exercice 4 - le savant fou en voyage**

Un chimiste souhaite transporter les produits $A, B, C, D, E$, dans des valises. Les couples de produits suivants ne doivent en aucun cas être transportés dans la même valise :
- $A$ et $C$
- $A$ et $D$
- $C$ et $D$
- $A$ et $B$
- $B$ et $D$
- $A$ et $E$
- $E$ et $C$

Le but est de répartir les produits dans un minimum de valises.

1. Ramenez-vous à un problème classique de théorie des graphes. Dessinez le graphe.
2. Ce problème est-il NP-Complet ?
3. Proposez une solution réalisable à trois valises pour ce problème.

**Exercice 5 - ensemble stable et $k$-coloration**

Soit $G = (S, A)$ un graphe, $k$ une constante strictement positive. On s'intéresse à la $k$-colorabilité du graphe $G$ (i.e. peut-on colorier le graphe $G$ avec $k$ couleurs ?). Nous allons créer un graphe $G' = (S', A')$ à partir de $G$ en procédant de la façon suivante :
- A chaque sommet $s$ de $G$, on associe $k$ sommets de $G'$, notés $\{(s, 1), \ldots, (s, k)\}$, tous reliés entre eux par des arêtes.
- Pour chaque arête $\{e, e'\}$ de $G$, et chaque couleur $j$, on relie dans $G'$ les sommets $(e, j)$ et $(e', j)$. Notez bien que comme les sommets sont des couples, les arêtes sont des **paires de couples**.

Plus formellement,
-
$$S' = \{(s, j) | s \in S, j \in \{1, \ldots, k\}\}$$
est un ensemble de **couples** (sommet de $G$/couleur).
-
$$A' = A_1 \cup A_2$$

avec

$$A_1 = \{\{(s,j),(s,j')\} | s \in S, 1 \leq j < j' \leq k\}$$

un ensemble d'arêtes formant $|S|$ sous-graphes complets de $k$ sommets et

$$A_2 = \{\{(e,j),(e',j)\} | \{e,e'\} \in A, j \in \{1,\ldots,k\}\}$$

un ensemble d'arêtes interconnectant ces sous-graphes.

Le but de cet exercice est de montrer qu'en trouvant un ensemble stable de $|S|$ sommets dans $G'$, on trouve aussi une $k$-coloration de $G$.

1. Soit $G$ le graphe de l'exercice précédent, on pose $k = 3$. Représentez graphiquement $G'$.
2. Donnez une définition en français (avec des mots) de ce qu'est un ensemble stable.
3. Ce problème (existe-t-il un ensemble stable de $|S|$ sommets dans $G'$ ?) est-il NP-Complet ?
4. Donnez un ensemble stable de 5 sommets dans ce graphe.
5. Utilisez cet ensemble stable pour déterminer une 3-coloration de $G$.
6. Démontrez de façon formelle et avec le plus de rigueur possible que, étant donné un graphe $G$ arbitraire, un nombre $k > 0$, $G$ est $k$-colorable si et seulement s'il existe un ensemble stable de $|S|$ sommets dans $G'$.

**Exercice 6 - Ordonnancement**

En votre qualité de manager, vous vous retrouvez confronté au problème suivant : vous avez un ensemble de tâches à effectuer, chacune d'elle ayant un temps d'exécution connu à l'avance et une date de fin souhaitée. De plus, il existe pour chaque tâche une date avant laquelle il n'est pas possible de commencer son exécution. Votre équipe ne peut pas travailler sur deux tâches différentes en même temps et il n'est pas possible de fractionner ou d'interrompre l'exécution d'une tâche. Si l'exécution d'une tâche n'est pas achevée avant la date de fin souhaitée, on dit de cette tâche qu'elle est en retard. Votre objectif est de déterminer un calendrier d'exécution des tâches de sorte à minimiser la somme des retards. Voici les projets à réaliser en septembre :

| numéro | nom | exécution au plus tôt | durée | deadline |
|---|---|---|---|---|
| 1 | changer la plomberie | 1 | 1 | 6 |
| 2 | débugger le projet $A$ | 2 | 3 | 4 |
| 3 | nettoyer la voiture du PDG | 1 | 1 | 6 |
| 4 | installer le progiciel $B$ | 8 | 2 | 23 |
| 5 | créer la base de donnée pour $C$ | 1 | 5 | 12 |
| 6 | coder une interface graphique pour $C$ | 10 | 5 | 15 |
| 7 | préparer le planning d'octobre | 17 | 2 | 19 |

Par exemple, l'installation du progiciel $B$ prend 2 jours consécutifs, elle ne peut pas commencer avant le 8 septembre, et doit se terminer au plus tard le 23 septembre. Il possible de commencer le 8 et de terminer le 9 tout comme il est possible de commencer le 22 et de terminer le 23.

1. Donnez une solution réalisable pour ce problème. Vous présenterez la solution sous forme d'un tableau, chaque tâche occupera une ligne et celles-ci seront disposées dans l'ordre de leur exécution. Vous utiliserez quatre colonnes
   - le nom de la tâche
   - la date de début
   - le temps d'exécution
   - la date de fin d'éxecution
2. Formalisons le problème.
   - **Données :** un nombre $n$ de tâches, un ensemble de durées $\{d_1,\ldots,d_n\}$, un ensemble de dates de début d'exécution au plus tôt $\{r_1,\ldots,r_n\}$, un ensemble de dates de fin d'exécution au plus tard $\{f_1,\ldots,f_n\}$, une constante $M$.
   - **Question :** Existe-t-il un ordre d'exécution des tâches tel que la somme cumulée des retards n'excède pas $M$ ?

   Nous appelerons ce problème SCHEDULING. Soient $\{s_1,\ldots,s_n\}$ les dates de début d'exécution de chacune des tâche. Soient $\{t_1,\ldots,t_n\}$ les dates de fin d'exécution. Donnez une relation simple entre $t_i$, $s_i$ et $d_i$. Prenez bien soin de vérifier que cette relation est vérifiée dans l'exemple de la question précédente.
3. L'exécution de la tâche $i$ ne peut pas commencer avant la date $r_i$. Exprimez cette contrainte avec les notations des questions précédentes.

4. Deux tâches d'indices respectifs $i$ et $j$ ne peuvent pas se superposer, exprimez cette contrainte à l'aide d'une proposition portant sur $s_i$, $s_j$, $t_i$ et $t_j$.
5. Combien y a-t-il de contraintes de cette forme ? Vous justifierez votre réponse en dénombrant les couples $(i, j)$ tels que $1 \leq i < j \leq n$.
6. Nous notons $\max(a, b)$ la plus grande des deux valeurs $a$ et $b$. A quoi correspond la valeur $\max(0, t_i - f_i)$ ?
7. Nous choisissons comme fonction objectif la somme cumulée des retards. Exprimez cette fonction objectif à partir des $t_i$ et des $f_i$
8. Exprimez la condition "la somme cumulée des retards n'excède pas $M$" avec les notations de la question précédente

**Exercice 7 - Sac-à-dos**

Nous souhaitons résoudre le problème du sac à dos par énumération exhaustive des solutions, c'est à dire comme des boeufs. Nous représenterons une solution $s$ (réalisable ou non) par la liste chaînée des indices des objets sélectionnés dans $s$. Nous représenterons une instance $i$ par deux tableaux $p$ (poids des objets) et $v$ (valeurs des objets). Une solution sera la liste chaînée des indices des objets placés dans le sac. Ecrivez les corps des sous-programmes suivants :

1. `void* linkedListCopy(void* l)`. Copie les maillons de la liste chaînée `l`, mais pas les valeurs pointées par les maillons. Vous utiliserez `linkedListMap`.
2. Ecrivez le sous-programme `void knapSackPrintSolution(linkedList* l)`, affiche la liste des indices formant la solution `l`, vous utiliserez `linkedListApply`.
3. Ecrivez un sous-programme `linkedList* parties(linkedList* l)`, ce sous-programme prend en paramètre un ensemble représenté par une liste chaînée et retourne une liste chaînée dans laquelle chaque maillon est une liste chaînée contenant une partie de $l$. Vous utiliserez `linkedListApply`.
4. `void knapSackPrintParties(linkedList* l)`, affiche toutes les parties de `l`, vous utiliserez `linkedListApply`.
5. `void knapSackDestroyParties(linkedList* l)` détruit `l` et les listes chaînées pointées par chaque maillon de cette liste.

Nous représenterons une instance du problème sac à dos avec une variable du type :

```
typedef struct
{
  /*
  nombre total d'objets
  */
  long nbItems;

  /*
  tableau de nombreObjets elements contenant le poids de
  chaque element
  */
  long* weights;

  /*
  tableau de nombreObjets elements contenant la valeur de
  chaque element
  */
  long* values;

  /*
  Poids maximum qu'il est possible de placer dans le sac à dos.
  */
  long maxWeight;
}knapSack;
```

Ecrivez les corps des sous-programmes suivants :

1. `knapSack* knapSackMake(long nbItems, long maxWeight)`, alloue la mémoire pour contenir l'instance ainsi que les deux tableaux.
2. `void knapSackSetItem(knapSack* k, long index, long value, long weight)`, fixe les poids (`weight`) et valeur (`value`) de l'objet d'indice `i` de l'instance `k`.
3. `long knapSackValue(knapSack* instance, linkedList* solution)`, retourne la somme des valeurs des objets de `instance` dont les indices sont dans la liste `solution`. Vous utiliserez la fonction `linkedListApply`.
4. `long knapSackWeight(knapSack* instance, linkedList* solution)`, retourne le poids de la solution `solution` relative à l'instance `instance`. Vous procéderez de la même façon que dans la question précédente.
5. `int knapSackIsFeasible(knapSack* instance, linkedList* solution)`, retourne vrai si est seulement si la solution `solution` relative à l'instance `instance` est réalisable.
6. `void knapSackPrintInstance(knapSack* k)`, affiche la liste des poids des des valeurs de tous les objets de l'instance `k`. Par exemple,

```
(i =   0, v =   1, w =   1)
(i =   1, v =   2, w =   2)
(i =   2, v =   9, w =   7)
(i =   3, v =   8, w =   2)
(i =   4, v =   5, w =   7)
(i =   5, v =   6, w =   6)
(i =   6, v =   7, w =   7)
(i =   7, v =   4, w =   2)
(i =   8, v =   3, w =   7)
(i =   9, v =  10, w =   2)
(i =  10, v =   1, w =   1)
(i =  11, v =   2, w =   2)
(i =  12, v =   9, w =   7)
(i =  13, v =   8, w =   2)
(i =  14, v =   5, w =   7)
(i =  15, v =   6, w =   6)
(i =  16, v =   7, w =   7)
(i =  17, v =   4, w =   2)
(i =  18, v =   3, w =   7)
(i =  19, v =  10, w =   2)
(i =  20, v =   1, w =   1)
```

7. `void knapSackDestroy(knapSack* k)`, vous avez vraiment besoin que je vous donne des indications ?
8. `linkedList* knapSackListInit(long n)`, retourne une liste chaînée contenant les éléments $\{1, \ldots, n\}$. Vous l'implémenterez de façon récursive.
9. `linkedList* knapSackBruteForceSolve(knapSack* k)` retourne la liste chaînée contenant la liste des indices des objets formant la solution optimale du problème `k`.
10. Jusqu'à quelle valeur de $n$ parvenez-vous à résoudre le problème ?
11. Montrez que le temps d'exécution est proportionnel au nombre de maillons crées, toutes listes confondues.
12. Evaluez le nombre de maillons en fonction de $n$, donnez la complexité de la résolution de knapsack par énumération de toutes les solutions.
13. Est-ce que cela vous explique pourquoi l'exécution est si longue ?

# Chapitre 9

# Programmation linéaire

## 9.1 Exemple

Nous disposons de 500 kilos de matière première $M_A$, et de 600 kilos de matière première $M_B$. Pour fabriquer un exemplaire de produit $P_1$ il faut 10 kilos de $M_A$ et 20 kilos de $M_B$, pour fabriquer un exemplaire de produit $P_2$ il faut 20 kilos de $M_A$ et 10 kilos de $M_B$. La vente d'un exemplaire de $P_1$ rapporte 10 euros, celle d'une exemplaire de $P_2$ rapporte 12 euros. Soit $x_1$ le nombre de produits $P_1$ fabriqués $x_2$ le nombre de produits $P_2$ fabriqués. Comment fixer $x_1$ et $x_2$ pour maximiser les bénéfices ?

Ce problème se modélise et se note de la façon suivante :

$$\begin{cases} max & 10x_1 & + & 12x_2 & & \\ s.c. & 10x_1 & + & 20x_2 & \leq & 500 \\ & 20x_1 & + & 10x_2 & \leq & 600 \\ & x_1, & & x_2 & \geq & 0 \end{cases}$$

## 9.2 Algorithme du simplexe en bref

Le but de ce cours n'est pas de vous apprendre comment on résoud les programmes linéaires, mais comment on les modélise. Ces problèmes se résolvent, en pratique, en temps polynomial. Un des algorithmes les plus connus est l'algorithme du simplexe. Je n'expliquerai ici que le principe, pour plus de détails, voir [5]. Les contraintes peuvent être représentées par des demi-plans. L'ensemble des solutions réalisables est l'intersection de ces demi-plans, cette intersection forme un polygone convexe. Soit $z$ un réel, l'ensemble des couples $(x, y)$ tel que la fonction objectif prend la valeur $z$ est une droite $d_z$.

En Augmentant la valeur de $z$, on "déplace" $d_z$. On cherche donc la plus grande valeur de $z$ telle qu'au moins un point de $d_z$ se trouve dans l'ensemble des solutions réalisables. Il se trouve qu'à l'optimum, la droite $d_z$ passe par un des sommets du polygone des solutions réalisables. L'algorithme du simplexe parcours les sommets de ce polygone jusqu'à atteindre cet optimum. Cette méthode se généralise facilement à un nombre de variables et de contraintes arbitraires.

## 9.3 Modéliser un programme linéaire

Un programme linéaire à $n$ variables et à $m$ contraintes est de la forme suivante :

$$\begin{cases} max & c_1x_1 & + & \dots & + & c_ix_i & + & \dots & + & c_nx_n & & \\ s.c. & a_{11}x_1 & + & \dots & + & a_{1i}x_i & + & \dots & + & a_{1n}x_n & \leq & b_1 \\ & \vdots & & & & \vdots & & & & \vdots & & \\ & a_{j1}x_1 & + & \dots & + & a_{ji}x_i & + & \dots & + & a_{jn}x_n & \leq & b_j \\ & \vdots & & & & \vdots & & & & \vdots & & \\ & a_{m1}x_1 & + & \dots & + & a_{mi}x_i & + & \dots & + & a_{mn}x_n & \leq & b_m \end{cases}$$

Soient $x$ le vecteur à $n$ lignes et 1 colonne

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$c$ le vecteur à $n$ lignes et 1 colonne

$$\begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$$

Alors on exprime la valeur de la fonction objectif avec le produit scalaire $c^t x$. Soit $A$ la matrice à $m$ lignes et $n$ colonnes

$$\begin{pmatrix} a_{11} & \dots & a_{1i} & \dots & a_{1n} \\ \vdots & & \vdots & & \vdots \\ a_{j1} & \dots & a_{ji} & \dots & a_{jn} \\ \vdots & & \vdots & & \vdots \\ a_{m1} & \dots & a_{mi} & \dots & a_{mn} \end{pmatrix}$$

et $b$ le vecteur à $m$ lignes et 1 colonne

$$\begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

Alors les contraintes s'écrivent

$$Ax \leq b$$

Finalement, un programme linéaire est de la forme

$$\begin{cases} max & c^t x \\ s.c. & Ax \leq b \end{cases}$$

**Exercice 1 - Problème de transport**

Une entreprise de fabrication de mort aux rat dispose de $n$ points de fabrication, produisant $p_i$ $(1 \leq i \leq n)$ tonnes par jour. Il faut acheminer dans $m$ points de vente les quantités $d_j$ $(1 \leq j \leq m)$. On a bien sûr

$$\sum_{i=1}^{n} p_i \geq \sum_{j=1}^{m} d_j$$

Le coût d'acheminement de 1 tonne de mort au rat du point de fabrication $i$ au point de vente $j$ est donné par $c(i, j)$. Comment faire pour satisfaire la demande de chaque point de vente tout minimisant le coût total d'acheminement ?

Modélisez ce problème par un programme linéaire.

**Exercice 2 - Minimiser le nombre de transactions**

$m$ personnes partent en vacances et paient de façon anarchique pendant leur séjour. A leur retour, ils font les comptes et le chargé des comptes dessine un graphe orienté $G = (S, A)$ muni d'une valuation $c$ où

– chaque élément de $S$ est une personne

– chaque couple $(i, j) \in A$ signifie la personne $i$ doit $c(i, j)$ euros à la personne $j$

Représentez le graphe correspondant aux donnés suivantes :

1. $A$ doit 147 euros à $C$

2. $B$ doit 57 euros à $A$
3. $E$ doit 228 euros à $B$
4. $B$ doit 129 euros à $D$
5. $D$ doit 78 euros à $A$
6. $C$ doit 45 euros à $B$
7. $B$ soit 187 euros à $E$

On remarque que ce graphe contient des circuits, et que pour certains couples, de sommets, il existe plusieurs chemins. Utiliser le graphe tel qu'il est pour planifier les remboursements conduirait à des remboursements redondants ($C$ donne à $B$ qui donne à $A$ qui donne à $C$...). En réfléchissant, il remarque que peu importe qui rembourse qui, ce qui compte, c'est que chacun retrouve son argent. Pour simplifier le problème, il décompose donc $S$ en deux sous-ensembles $S^+$ (ceux à qui on doit de l'argent qu'ils n'en doive) et $S^-$ (ceux qui doivent de l'argent plus qu'on leur en doit). Chaque sommet $s$ est étiqueté par la valeur

$$|\sum_{t\in\Gamma^+(s)} c(s,t) - \sum_{t\in\Gamma^-(s)} c(t,s)|$$

Quels sont les sommets et les étiquettes des sommets de ce nouveau graphe ? La question est de trouver comment placer les arêtes de la façon la plus simple possible. Modélisez ce problème comme un problème de flot max. Nous souhaitons maintenant minimiser le nombre de transaction. Est-ce possible avec un algorithme de flot maximum ? Nous verrons dans une autre section que ce problème est NP-Complet.

**Exercice 3 - Plaques d'égoût**

Une entreprise de fabrication de plaques d'égoûts dispose de $n$ points de fabrication, produisant $p_i$ $(1 \leq i \leq n)$ plaques par jour. Il faut chaque jour acheminer dans $m$ centres les quantités $d_j$ $(1 \leq j \leq m)$. Toutes les distances sont suffisament courtes pour que chaque trajet puisse être fait par un seul camion dans une même journée. La production est faite telle que

$$\sum_{i=1}^{n} p_i = \sum_{j=1}^{m} d_j$$

Chaque camion à une contenance $K$, le problème est comment affecter chaque camion à un centre de fabrication et un centre de livraison pour utiliser un minimum de camions ? Montrez que ce problème est le même que le précédent.

**Exercice 4 - Regression linéaire et écart-absolu**

Soit une série statistique bivariée $\{(x_i, y_i)|i = 1, \ldots, n\}$, une regression linéaire consiste en la recherche de la droite passant au plus près des points représentés dans un repère orthonormé à deux dimensions.

Soit $y = ax + b$ l'équation de cette droite. Alors, on approxime $y_i$ avec la valeur $ax_i + b$. L'écart absolu entre $y_i$ et son approximation est $|ax_i + b - y_i|$. Nous souhaitons minimiser la somme es écarts absolus

$$\sum_{i=1}^{n} |ax_i + b - y_i|$$

Modélisez ce problème comme un problème de programmation linéaire.

## 9.4 GLPK

### 9.4.1 Qu'est-ce que c'est ?

GLPK [6] est une librairie de résolution de programmes linéaires. Elle contient une implémentation de l'algorithme du simplexe. Il est possible de lancer GLPK en ligne de commande, ou bien d'importer la librairie dans un programme C.

### 9.4.2 Où le trouver ?

Utilisez urpmi ou apt-get.

### 9.4.3 La documentation

Le manuel de référence[7] est très complet mais quelque peu difficile à lire quand on n'est pas habitué. Cette annexe est aussi là pour vous vous aider à le lire.

### 9.4.4 Les fonctions

#### Exemple

Nous allons illustrer le fonctionnement de GLPK en résolvant le problème

$$\begin{cases} max & 10x_1 & + & 12x_2 & & \\ s.c. & 10x_1 & + & 20x_2 & \leq & 500 \\ & 20x_1 & + & 10x_2 & \leq & 600 \\ & x_1, & & x_2 & \geq & 0 \end{cases}$$

Le programme C ci-dessous modélise ce problème et l'envoie dans glpk.

```
#include<stdio.h>
#include"glpk.h"

int main()
{
  LPX* lp = lpx_create_prob();
  double coeffs[4+1];
  int rows[4+1], cols[4+1];
  lpx_set_obj_dir(lp, LPX_MAX);
  lpx_add_rows(lp, 2);
  lpx_set_row_bnds(lp, 1, LPX_UP, 0.0, 500.0);
  lpx_set_row_bnds(lp, 2, LPX_UP, 0.0, 600.0);
  lpx_add_cols(lp, 2);
  lpx_set_col_bnds(lp, 1, LPX_LO, 0.0, 0.0);
  lpx_set_col_bnds(lp, 2, LPX_LO, 0.0, 0.0);
  lpx_set_obj_coef(lp, 1, 10.);
  lpx_set_obj_coef(lp, 2, 12.);
  coeffs[1] = 10 ; rows[1] = 1 ; cols[1] = 1 ;
  coeffs[2] = 20 ; rows[2] = 1 ; cols[2] = 2 ;
  coeffs[3] = 20 ; rows[3] = 2 ; cols[3] = 1 ;
  coeffs[4] = 10 ; rows[4] = 2 ; cols[4] = 2 ;
  lpx_load_matrix(lp, 4, rows, cols, coeffs);
  lpx_simplex(lp);
  printf("valeur de la fonction objetif : %lf\n", lpx_get_obj_val(lp));
  printf("quantite de produits A : %lf\n", lpx_get_col_prim(lp, 1));
  printf("quantite de produits B : %lf\n", lpx_get_col_prim(lp, 2));
  lpx_delete_prob(lp);
  return 1;
}
```

N'oubliez pas de compiler vos sources avec l'option `-lglpk` (sinon, ça marche pas).

#### Importation

Avant toute chose, n'oubliez pas

```
#include ''glpk.h''
```

### Création, suppression

GLPK met à votre disposition le type `LPX`, une variable de ce type est un programme linéaire. On crée un programme linéaire avec la fonction `lpx_create_prob()`, on le détruit avec `lpx_delete_prob(LPX*)`.

### Nombre de variables et de contraintes

Les programmes linéaires sont usuellement représentés par des tableaux :

$$\left\{\begin{array}{llllllllllll} max & c_1x_1 & + & \ldots & + & c_ix_i & + & \ldots & + & c_nx_n & & \\ s.c. & a_{11}x_1 & + & \ldots & + & a_{1i}x_i & + & \ldots & + & a_{1n}x_n & \leq & b_1 \\ & \vdots & & & & \vdots & & & & \vdots & & \\ & a_{j1}x_1 & + & \ldots & + & a_{ji}x_i & + & \ldots & + & a_{jn}x_n & \leq & b_j \\ & \vdots & & & & \vdots & & & & \vdots & & \\ & a_{m1}x_1 & + & \ldots & + & a_{mi}x_i & + & \ldots & + & a_{mn}x_n & \leq & b_m \end{array}\right.$$

Vous remarquez que ce programme à $n$ variables et à $m$ contraintes est formé d'un grand tableau, à $n+1$ colonnes, et à $m+1$ lignes. Dans la terminologie de GLPK,

- Une variable est une colonne (`col`)
- Une contrainte est une ligne (`row`)

La fonction `lpx_add_rows(LPX*, int)` spécifie le nombre de contraintes, `lpx_add_cols(LPX*, int)` spécifie le nombre de variables.

### La fonction objectif

Soyez très atentifs : **les indices sous GLPK commençent à 1**. La fonction `lpx_set_obj_coef(LPX*, int numeroDeLigne, double coefficient)` permet de préciser la valeur d'un coefficient de la fonction objectif. Par exemple, pour affecter la valeur 3.0 à $c_2$ dans le programme *mylp*, on utilise l'instruction `lpx_set_obj_coef(mylp, 2, 3.0)`. Selon le contexte, on cherche à minimiser ou à maximiser la fonction objectif, on le précise avec la fonction `lpx_set_obj_dir(LPX*, int direction)`. Il existe deux directions : `LPX_MIN` et `LPX_MAX`.

### Le deuxième membre de l'inégalité

Il n'est sujet dans cette section que du membre de droite des inégalités formant les contraintes, nous verrons dans une partie ultérieure comment définir le membre de gauche de chaque contrainte. Il existe plusieurs formes de contraintes, par exemple,

$$a_{j1}x_1 + \ldots + a_{ji}x_i + \ldots + a_{jn} \leq b_j$$

Cette forme se note `LPX_UP` (upper). Une contrainte de la forme

$$a_{j1}x_1 + \ldots + a_{ji}x_i + \ldots + a_{jn}x_n \geq b_j$$

se note `LPX_LOW` (lower). Pour d'autres formes, reportez-vous à la documentation. On spécifie la forme d'une contrainte avec la fonction `lpx_set_row_bnds(LPX*, int indiceLigne, int forme, double borneInf, double borneSup)`. Par exemple, la forme de la contrainte

$$a_{41}x_1 + \ldots + a_{4i}x_i + \ldots + a_{4n}x_n \leq 67$$

se spécifie avec l'instruction `lpx_set_row_bnds(mylp, 4, LPX_UP, 0.0, 67.0)`. La forme de la contrainte

$$a_{41}x_1 + \ldots + a_{4i}x_i + \ldots + a_{4n}x_n \geq 67$$

se spécifie avec l'instruction `lpx_set_row_bnds(mylp, 4, LPX_LO, 67.0, 0.0)`. Notez que certains paramètres sont ignorés, le quatrième si la contrainte est de la forme `LPX_UP`, le cinquième si la contrainte est de la forme `LPX_LO`.

**Les domaines des variables**

Il est possible d'ajouter des contraintes sur les valeurs des variables. Par exemple, les contraintes suivantes

$$x_1, x_2 \geq 0$$

font partie du programme linéaire donné en exemple. La fonction `lpx_set_col_bnds(LPX*, int indiceColonne, int forme, double borneInf, double borneSup)`. Par exemple, si $x$ est d'indice 4, et qu'on tient à l'empêcher de prendre des valeurs négatives, on le spécifie avec l'instruction `lpx_set_col_bnds(mylp, 4, LPX_LO, 0.0, 0.0)`.

**Les coefficients des contraintes**

Les coefficients des contraintes sont tous initialisés à 0. Si vous tenez à ce que certains prennent des valeurs non nulles, vous devez le spécifier avec la fonction `lpx_load_matrix(LPX*, int nbCoeffs, int* lignesDesCoefficients, int* colonnesDesCoefficients, double* coefficients);`. Cette fontion prend en paramètre la taille des trois tableaux $lignesDesCoefficients$, $colonnesDesCoefficients$ et $coefficients$. Le $i$-ème élément de $coefficients$ est le coefficient de la contrainte d'indice $colonnesDesCoefficients[i]$ et de la ligne d'indice $lignesDesCoefficients[i]$. Par exemple, pour spécifier l'unique contrainte du programme linéaire

$$\begin{cases} max & x_1 & + & x_2 & & \\ s.c. & 5x_1 & + & 4x_2 & \leq & 3 \\ & x_1, & & x_2 & \geq & 0 \end{cases}$$

on initialise les tableaux de la sorte

```
coefficients[1] = 5 ;
        lignesDesCoefficients[1] = 1 ;
        colonnesDesCoefficients[1] = 1
coefficients[2] = 4 ;
        lignesDesCoefficients[2] = 1 ;
        colonnesDesCoefficients[2] = 2
```

Ensuite, il n'y a plus qu'à invoquer

```
lpx_load_matrix(mylp, 2, lignesDesCoefficients,
                colonnesDesCoefficients, coefficients);
```

**L'algorithme du simplexe**

`lpx_simplex(LPX*)` exécute l'algorithme du simplexe sur le problème passé en paramètre. La fonction `double lpx_get_obj_value(LPX*)` retourne la valeur de la fonction objectif à l'optimum. `double lpx_get_col_primm(LPX*, int indiceVariable)` retourne la valeur de la $indiceVariable$-ème variable à l'optimum.

**Exercice 5 - Prise en main de GLPK**

Utilisez GLPK pour résoudre le programme linéaire :

$$\begin{cases} max & 4x_1 & + & 6x_2 & + & 2x_3 & & \\ s.c. & 3x_1 & + & 7x_2 & + & x_3 & \leq & 12 \\ & \frac{3}{5}x_1 & + & 3x_2 & + & \frac{14}{5}x_3 & \leq & 3 \\ & 9x_1 & + & \frac{9}{100}x_2 & + & 5x_3 & \leq & 6 \\ & x_1, & & x_2, & & x_3 & \geq & 0 \end{cases}$$

**Exercice 6 - Le plus beau métier du monde**

Un enseignant a donné $k$ notes de contrôle continu à ses $n$ élèves, et une note de partiel pour chaque élève. On note $c_i^j$ la note obtenue par le $i$-ème élève au $j$-ème contrôle continu, et $p_i$ la note de partiel du $i$-ème élève. Il souhaite déterminer le jeu de coefficients qui maximise la moyenne générale de la classe.

1. Comment résoudre ce problème ? Utilisez GLPK pour résoudre l'instance

```
double notesPartiel[N] = {10, 13,  2, 4.5,  9, 12, 16,  8, 5, 12};
double notesCC1[N] =     {11, 16, 11,   8, 10, 11, 11,  6, 4, 13};
double notesCC2[N] =     { 9, 18,  7,   8, 11, 3,  13, 10, 7, 19};
```

2. Que remarquez-vous ?
3. Le responsable pédagogique est en total désaccord, il considère que la note de partiel doit avoir un coefficient au moins égal à 0.5 et que deux notes de contrôle continu doivent avoir des coefficients au moins égaux à 0.1. Comment tenir compte de ces contraintes ?
4. En raison d'un petit nombre de notes très élevées à certains examens, la moyenne générale de la classe est bonne malgré le fait que beaucoup d'élèves ont une moyenne très basse. L'enseignant décide donc de changer de critère et de maximiser la moyenne la plus basse. Comment procéder ?
5. Vérifiez la solution de la question précédente en calculant les moyennes de tous les élèves avec les coefficients donnés par GLPK (ne le calculez pas à la main, programmez-le...).

# Chapitre 10

# Théorie de la complexité

Certains problèmes combinatoires sont dits NP-Complets, cela signifie qu'il n'est possible de les résoudre qu'en passant par une recherche exhaustive dans l'ensemble des solutions, ce qui est concrètement irréalisable. Le but de ce cours est de vous montrer comment reconnaître qu'un problème est NP-Complets, et comment le prouver.

Le chapitre de [2] sur la NP-Complétude fournit un exposé clair de la théorie de la complexité, mais utilisant des concepts de calculabilité que je n'utilise pas dans ce cours. La lecture de ce chapitre ne pourra cependant que vous être bénéfique. L'ouvrage [8] est la bible de la théorie de la complexité. Bien qu'il ne soit disponible qu'en anglais et relativement difficile à se procurer, la théorie y est expliquée de façon claire, détaillée, et sans excès de formalisme.

## 10.1 Terminologie et rappels

### 10.1.1 Problèmes de décision

Nous nous restreindrons aux **problèmes de décision** dans le reste du cours. Un problème de décision est spécifié par

- Des données
- Une question

Par exemple,

- Données : un graphe $G$, une constante $k$ strictement positive
- Question : $G$ est-il $k$-colorable ?

### 10.1.2 Instances

Une **instance** d'un problème s'obtient en précisant les valeurs des données. Par exemple,

- Données : un graphe $G$ complet d'ordre 10, $k = 9$
- Question : $G$ est-il $k$-colorable ?

On résout un problème de décision et **décidant**, pour toute instance $I$, si cette instance est à réponse **oui** ou à réponse **non**. On dit plus généralement qu'un algorithme qui résout un problème de décision **décide** ce problème, et toutes les instances de ce problème.

### 10.1.3 Algorithme polynomial

Un algorithme est polynomial si sa complexité est de l'ordre de $\mathcal{O}(n^k)$ où $n$ est la taille du problème, et $k$ une **constante** indépendante de l'instance considérée.

## 10.2 Les classes P et NP

La théorie de la complexité est basée sur une classification des différents problèmes. Deux classes sont importantes à connaître : $P$ et $NP$.

### 10.2.1 La classe P

Un problème appartient à la classe de problèmes $P$ s'il existe un algorithme polynomial permettant de le résoudre. Par exemple, le plus court chemin, l'arbre couvrant dans un graphe, etc.

### 10.2.2 La classe NP

Etant donné un problème de décision, ce problème est dans NP s'il est possible de vérifier une solution en temps polynomial. Considérons par exemple le problème suivant :

- Données : un graphe $G$
- Question : existe-t-il un cycle hamiltonien dans $G$ ?

Supposons que ce graphe contienne 100 sommets, et qu'un de vos camarade vous dise : "cette instance est à réponse oui". Vous lui demanderez certainement : "dans quel ordre parcourir les sommets pour former un cycle hamiltonien" ? Notez bien que malgré le fait que résoudre le problème soit infaisable en pratique, vérifier qu'un chemin est effectivement un chemin hamiltonien est très aisé. Un cycle permettant de vérifier en temps polynomial qu'une telle instance est une instance à réponse oui s'appelle un **certificat**.

Plus généralement, étant donné une instance $I$ d'un problème $\Pi$, un certificat $c$ est une donnée permettant de vérifier en temps polynomial que $I$ est une instance de $\Pi$ à réponse oui.

### 10.2.3 Relations d'inclusion

Tout problème de $P$ est aussi un problème de $NP$. Ce qui s'écrit aussi

**Théorème 10.2.1** $P \subset NP$

Soit $I$ une instance d'un problème $\Pi$ appartenant à $P$, alors il existe un algorithme polynomial permettant de décider $\Pi$. De ce fait, si $I$ est à réponse oui, même un certificat vide permet de le vérifier en temps polynomial.

La question suivante empêche la plupart des chercheurs en informatique de dormir : Est-ce que $P = NP$ ? On sait déjà que $P \subset NP$, pour répondre à la question, il faudrait montrer :

- $NP \subset P$, autrement dit, tout problème de $NP$ peut être résolu en temps polynomial.
- $NP \setminus P \neq \emptyset$, autrement dit, il existe des problèmes de $NP$ qui ne peuvent être résolus en temps polynomial.

Notez que le problème du cycle hamiltonien appartient à $NP$.

- Si $P = NP$, alors il existe un algorithme polynomial permettant de le décider.
- Si $P \neq NP$, alors il convient de se demander si ce problème n'est pas dans $NP \setminus P$, le cas échéant il n'existe aucun algorithme polynomial permettant de le décider.

Etant donné le nombre de problèmes de $NP$ pour lesquels on ne trouve aucun algorithme polynomial, il est fort probable que $P \neq NP$. Notez bien que ceci n'est qu'une conjecture, elle n'est pas démontrée à ce jour.

## 10.3 Réductions polynomiales

Soit $\Pi$ un problème de $NP$ pour lequel vous ne parvenez pas à trouver d'algorithme polynomial, vous nous pouvez pas dire que ce problème n'est pas dans $P$, car vous ne savez pas si $NP \setminus P$ est vide ou non. Il va donc falloir procéder autrement. Nous allons introduire la notion de difficulté d'un problème, ainsi qu'un moyen de comparer des problèmes.

### 10.3.1 Définition

Nous allons établir une relation d'ordre sur les problèmes, nous la noterons

$$\Pi \leq \Pi'$$

ce qui signifie "$\Pi$ est moins difficile que $\Pi'$". $\leq$ est définie comme suit

**Définition 10.3.1** $\Pi \leq \Pi'$ *s'il existe une fonction $f$ de complexité polynomiale qui à toute instance $I$ de $\Pi$ associe une instance $I'$ de $\Pi'$ telle que $I$ est à réponse oui si et seulement si $I'$ est à réponse oui.*

### 10.3.2 Exemple

Montrons que le problème $HC$ (cycle hamiltonien) est moins difficile que $TSP$ (voyageur de commerce). Pour cela, construisons une fonction $f$ prenant en paramètre une instance du problème $HC$, et créant une instance $I$ du problème $TSP$. Il est très important de déterminer $f$ de sorte que $I$ soit à réponse oui **si et seulement si** $f(I)$ est à réponse oui. Rappelons la définition de $HC$ dans un graphe non orienté :

- Données : un graphe $G$ non orienté
- Question : existe-t-il un cycle hamiltonien dans $G$ ?

Ainsi que la définition de $TSP$,

- Données : un graphe $G'$ non orienté muni d'une valuation $c$, une constante $K$
- Question : existe-t-il dans un $G$ un cycle hamiltonien de longueur inférieure ou égale à $K$ ?

Nous définissons $f$ de la sorte, $f$ prend en paramètre les données de $HC$, à savoir le graphe $G = (S, A)$, on définit l'instance $f(I)$ de la sorte

- $G' = (S, A')$, $A'$ contient toutes les paires de sommets qu'il est possible de former avec les sommets de $G$. Autrement dit, $G'$ est un graphe complet.
- $c(e) = 1$ si $e \in A$, $c(e) = 2$ si $e \notin A$. Autrement dit, les arêtes de $A'$ ont la valeur 1 si elle se trouvaient dans $A$, et toutes les arêtes qui ont été ajoutées pour obtenir un graphe complet ont la valeur 2.
- $K = |S|$.

Il est maintenant nécessaire de montrer que $I$ est une instance à réponse oui si et seulement si $f(I)$ est une instance à réponse oui.

- Soit $I$ une instance de $HC$ à réponse oui. Alors il existe un cycle hamiltonien dans $G$. La longueur de ce cycle dans $G$ est $|S|$, à savoir le nombre de sommets. La longueur de ce cycle est aussi $|S|$ dans $G'$, car les arêtes empruntées dans $G'$ étaient toutes de valuation 1.
- Soit $I$ une instance de $HC$ telle que $f(I)$ est à réponse oui. Comme $f(I)$ est à réponse oui, alors il existe un cycle hamiltonien de longueur $K = |S|$. Comme ce cycle passe par $|S|$ arêtes, et que toutes les arêtes de $G'$ sont de valuation 1 ou 2, alors ce cycle n'emprunte que des arêtes de valuation 1. Comme les arêtes de valuation 1 de $G'$ sont aussi des arêtes de $G$, alors il existe un cycle hamiltonien dans $G$.

### 10.3.3 Application

**Propriété 10.3.1** *Si* $\Pi \leq \Pi'$ *et* $\Pi' \in P$, *alors* $\Pi \in P$.

Soit $I$ une instance de $\Pi$, comme $\Pi \leq \Pi'$, alors il existe $f$ polynomiale telle que $I$ est à réponse oui si et seulement si $f(I)$ est à réponse oui. Comme $\Pi' \in P$, alors $\Pi'$ est décidé par un algorithme polynomial. En appliquant cet algorithme à $f(I)$, on décide $I$. L'algorithme qu'on obtient est donc

- Déterminer $f(I)$ (polynomial)
- Décider $f(I)$ (polynomial)

Comme les deux algorithmes considérés sont polynomiaux, alors $I$ est décidé en temps polynomial. Donc $\Pi \in P$.

Il est important de retenir que $\Pi \leq \Pi'$ implique que l'existence d'un algorithme polynomial décidant $\Pi'$ implique l'existence d'un algorithme polynomial décidant $\Pi$.

## 10.4 Les problèmes NP-complets

### 10.4.1 Courte méditation

Si vous vous trouvez face à un problème $\Pi$ que vous ne parvenez pas à décider à l'aide d'un algorithme polynomial, il vous suffit de montrer, par exemple, que $TSP \leq \Pi$ pour ne pas vous faire mépriser par vos supérieurs. Supposons qu'il existe un algorithme polynomial décidant $\Pi$, alors vous en déduiriez immédiatement un algorithme décidant le problème $TSP$, qui tient tout le monde en échec... Vous montrerez de ce fait non pas que le problème $\Pi$ n'a pas de solution, mais que ni vous, ni personne d'autre (encore moins vos supérieurs...), n'est capable, pour le moment, de trouver un algorithme polynomial pour le décider.

### 10.4.2 3-SAT

La question que nous sommes amenés à nous poser maintenant est : existe-t-il un problème de $NP$ au moins aussi difficile que tous les problèmes de $NP$ ? La réponse est oui.

**Théorème 10.4.1** *(Cook) Pour tout* $\Pi \in NP$, $\Pi \leq 3-SAT$

Le problème $SAT$ est défini dans la problèmethèque. $3-SAT$ correspond au cas particulier dans lequel toutes les clauses contiennent 3 littéraux. $3-SAT$ est plus difficile que tous les problèmes de $NP$, cela signifie que s'il était possible de décider $3-SAT$ en temps polynomial, il serait alors possible de décider n'importe quel problème de $NP$ en temps polynomial.

### 10.4.3 Conséquences déprimantes

**Propriété 10.4.1** *Si* $\Pi \leq \Pi'$ *et* $\Pi' \leq \Pi''$, *alors* $\Pi \leq \Pi''$.

Soit $I$ une instance de $\Pi$, comme $\Pi \leq \Pi'$, alors il existe $f$ polynomiale telle que $I$ est à réponse oui si et seulement si $f(I)$ est à réponse oui. Comme $\Pi' \leq \Pi''$, alors il existe $g$ polynomiale telle que $f(I)$ est à réponse oui si et seulement si $(g \circ f)(I)$ est à réponse oui. Donc la réduction $(g \circ f)$ est polynomiale et $I$ est à réponse oui si et seulement si $(g \circ f)(I)$ est à réponse oui.

**Définition 10.4.1** $\Pi$ *est NP-Complet si pour tout problème* $\Pi' \in NP$, *on a* $\Pi' \leq \Pi$

Autrement dit, $\Pi$ est NP-Complet s'il est plus difficile que n'importe quel problème de $NP$.

**Propriété 10.4.2** *Si* $3-SAT \leq \Pi$, *alors* $\Pi$ *est NP-Complet.*

Pour tout $\Pi' \in NP$, $\Pi' \leq 3-SAT$ et $3-SAT \leq \Pi$, donc $\Pi' \leq \Pi$. Alors pour tout $\Pi' \in NP$, $\Pi' \leq \Pi$, donc $\Pi$ est NP-Complet.

**Propriété 10.4.3** *Si* $\Pi$ *est NP-Complet, et* $\Pi \leq \Pi'$, *alors* $\Pi'$ *est NP-Complet.*

Si $\Pi$ est NP-Complet, alors pour tout problème $\Pi'' \in NP$, $\Pi'' \leq \Pi$. Comme de plus $\Pi \leq \Pi'$, alors pour tout problème $\Pi'' \in NP$, $\Pi'' \leq \Pi'$. Donc $\Pi'$ est NP-Complet.

Si vous voulez montrer qu'un problème est NP-Complet, il suffit de montrer qu'il est plus difficile qu'un problème déjà connu comme étant NP-Complet.

### 10.4.4 Un espoir vain

**Propriété 10.4.4** *Si* $\Pi$ *et* $\Pi'$ *sont NP-Complets, et qu'il est possible de décider* $\Pi$ *en temps polynomial, alors on peut décider* $\Pi'$ *en temps polynomial.*

Comme $\Pi'$ appartient à $NP$, alors $\Pi' \leq 3-SAT$, et comme $\Pi$ est NP-Complet, alors $3-SAT \leq \Pi$. Donc $\Pi' \leq \Pi$. Si $\Pi \in P$, alors $\Pi' \in P$.

Cette propriété est très importante, elle signifie que si on parvenait à décider un seul problème NP-Complet en temps polynomial, alors on pourrait tous les décider en temps polynomial et on aurait $P = NP$.

**Exercice 1 - 3-SAT/CLIQUE**

3-SAT est défini de la façon suivante :

- Données : Un ensemble $\{C_j | j = 1, \ldots, p\}$ de clauses, un ensemble $\{x_i | i = 1, \ldots, n\}$ de variables. Chaque clause $C_j$ contient 3 littéraux $\{l_j^1, l_j^2, l_j^3\}$ pouvant être de la forme $x_i$ ou $\neg x_i$, ce que l'on notera avec une égalité pour simplifier les notations.
- Question : Existe-t-il une valuation des variables satisfaisant au moins un littéral de chaque clause ?

Une clique est un sous-graphe complet. La taille d'une clique est le nombre de sommets qu'elle contient. On définit le problème CLIQUE de la façon suivante :

- Données : Une graphe $G$ non orienté, une constante $K > 0$
- Question : Existe-t-il dans $G$ une clique de taille $K$ ?

Nous considérons la réduction suivante :

- $K = p$
- A chaque clause $C_i$, on associe trois sommets $s_j^1, s_j^2$ et $s_j^3$.
- Il existe une arête entre deux sommets $s_j^i$ et $s_{j'}^{i'}$ si

– $j \neq j'$
– si $l_i^j = x$, alors $l_{j'}^{i'} \neq \neg x$

3-SAT est connu pour être NP-Complet, montrez que CLIQUE est NP-Complet.

**Exercice 2 - K-COLORATION/ENSEMBLE STABLE**

Soit $G = (S, A)$ un graphe, $k$ une constante strictement positive. On s'intéresse à la $k$-colorabilité du graphe $G$ (i.e. peut-on colorier le graphe $G$ avec $k$ couleurs ?). Nous allons créer un graphe $G' = (S', A')$ à partir de $G$ en procédant de la façon suivante :

– A chaque sommet $s$ de $G$, on associe $k$ sommets de $G'$, notés $\{(s,1), \ldots, (s,k)\}$, tous reliés entre eux par des arêtes.
– Pour chaque arête $\{e, e'\}$ de $G$, et chaque couleur $j$, on relie dans $G'$ les sommets $(e, j)$ et $(e', j)$. Notez bien que comme les sommets sont des couples, les arêtes sont des **paires de couples**.

Plus formellement,

–
$$S' = \{(s,j) | s \in S, j \in \{1, \ldots, k\}\}$$
est un ensemble de **couples** (sommet de $G$/couleur).
–
$$A' = A_1 \cup A_2$$
avec
$$A_1 = \{\{(s,j),(s,j')\} | s \in S, 1 \leq j < j' \leq k\}$$
un ensemble d'arêtes formant $|S|$ sous-graphes complets de $k$ sommets et
$$A_2 = \{\{(e,j),(e',j)\} | \{e,e'\} \in A, j \in \{1, \ldots, k\}\}$$
un ensemble d'arêtes reliant ces sous-graphes.

Démontrez de façon formelle et avec le plus de rigueur possible que, étant donné un graphe $G$ arbitraire, un nombre $k > 0$, $G$ est $k$-colorable si et seulement s'il existe un ensemble stable de $|S|$ sommets dans $G'$.

**Exercice 3 - CLIQUE/ENSEMBLE STABLE**

Montrez que ENSEMBLE STABLE est NP-Complet, vous tiendrez compte du fait que CLIQUE est NP-Complet.

**Exercice 4 - ENSEMBLE STABLE/COUVERTURE**

Montrez que COUVERTURE est NP-Complet, vous tiendrez compte du fait que ENSEMBLE STABLE est NP-Complet.

**Exercice 5 - 3-SAT/SUBSET SUM**

Le problème SUBSET SUM, est défini comme suit :

– Données : un ensemble $A = \{p_i | i \in \{1, \ldots, n\}\}$ de $n$ valeurs, une constante $K$.
– Question : existe-t-il un sous-ensemble $I$ de $\{1, \ldots, n\}$ tel que
$$\Sigma_{i \in I} p_i = K ?$$

On peut montrer par une réduction de $3-SAT$ que $SUBSETSUM$ est NP-Complet. Effectuons la transformation suivante :

– A chaque variable $x_j$, on associe deux valeurs $v_j$ et $v'_j$ dont la représentation décimale comporte $p + n$ chiffres.
  – Les $p$ premiers chiffres représentent les clauses. Le $i$-ème chiffre ($\in \{1, \ldots, p\}$) de $v_j$ prend la valeur 1 si $x_j$ apparaît dans $C_i$, 0 sinon. Le $i$-ème chiffre ($\in \{1, \ldots, p\}$) de $v'_j$ prend la valeur 1 si $\neg x$ apparaît dans $C_i$, 0 sinon.
  – Les $n$ derniers chiffres représentent l'indice de la variable $x_i$, le $p + j$-ème chiffre de $v_j$ et de $v'_j$ prend la valeur 1, les $n - 1$ autres chiffres prennent la valeur 0.
– A chaque clause $C_i$, on associe 2 valeurs $x_i$ et $x'_i$ de $n + p$ chiffres dont le $i$-ème chiffre vaut 1 et tous les autres 0.
– On prend comme nombre $K$ la valeur dont les $p$ premiers chiffres valent 3 et les $n$ derniers 1.

Utilisez la réduction polynomiale ci-dessus pour montrer que SUBSET SUM est NP-Complet.

**Exercice 6 - PARTITION/SUBSET SUM**

Le problème PARTITION est défini comme suit :
- Données : un ensemble $A = \{p_i | i \in \{1, \ldots, n\}\}$ de $n$ valeurs, tel que

$$\Sigma_{i=1}^{n} p_i$$

est pair.
- Question : existe-t-il un sous-ensemble $I$ de $\{1, \ldots, n\}$ tel que

$$\Sigma_{i \in I} p_i = \Sigma_{i \notin I} p_i$$

Montrez que SUBSET SUM est NP-Complet, utilisez le fait que PARTITION est NP-Complet.

**Exercice 7 - PARTITION/SAC A DOS**

Montrez, par réduction du problème PARTITION, que SAC A DOS est NP-Complet.

**Exercice 8 - Ordonnancement**

1. Montrez que SCHEDULING $\in$ NP.
2. Etant donné le problème PARTITION :
   - **Données :** Un ensemble de $m$ valeurs $\{v_1, \ldots, v_m\}$.
   - **Question :** Existe-t-il un sous-ensemble $I$ de $\{1, \ldots, m\}$ tel que $\sum_{i \in I} v_i = \sum_{i \notin I} v_i$

   Autrement dit, est-il possible de partitionner l'ensemble $\{v_1, \ldots, v_m\}$ en deux sous-ensembles dont la somme des valeurs est égale ? L'instance de PARTITION suivante, $m = 7$, $(v_1, \ldots, v_7) = (1, 2, 3, 4, 5, 6, 7)$ est-elle à réponse "oui" ? Vous démontrerez la justesse de votre réponse.
3. Nous savons que PARTITION est NP-Complet. Nous allons montrer à l'aide d'une réduction polynomiale $f$, que PARTITION $\leq$ SCHEDULING. Nous construisons une fonction $f$ qui à toute instance de PARTITION associe une instance de SCHEDULING définie comme suit :
   - $n = m + 1$
   - pour tout $i \in \{1, \ldots, m\}$, $d_i = v_i$, $r_i = 1$, $f_i = \sum_{j=1}^{n} v_j + 1$
   - $d_n = 1$, $r_n = \frac{1}{2} \sum_{j=1}^{n} v_j + 1$, $f_n = \frac{1}{2} \sum_{j=1}^{n} v_j + 1$
   - $M = 0$

   Déterminer l'image de l'instance $(1, 2, 3, 4, 5, 6, 7)$ par la fonction $f$. Vous présenterez votre solution sous forme d'un tableau, vous préciserez les valeurs de toutes les données du problème.
4. L'image de cette instance est-elle à réponse "oui" ? Vous démontrerez la justesse de votre réponse en donnant les valeurs de début d'exécution si c'est le cas, ou bien une preuve d'impossibilité dans le cas contraire.
5. Montrez, sans excès de formalisme, que $f$ est polynomiale.
6. Montrer que toute instance de PARTITION à réponse oui a une image par $f$ à réponse oui.
7. Montrer que toute instance de PARTITION dont l'image est à réponse oui est à réponse oui.
8. SCHEDULING est-il NP-Complet ? **Justifiez votre réponse**.

# Chapitre 11

# Introduction la cryptographie

## 11.1 Définitions

**Chiffrer** une information revient à la transformer de sorte qu'elle ne puisse être intelligible que par la personne à qui elle est destinée. La **cryptologie** est un terme générique regroupant deux disciplines :

- La **cryptographie**, qui est l'étude des méthodes de **chiffrement** et de **déchiffrement**.
- La **cryptanalyse**, qui est l'art de casser des messages chiffrés. Autrement dit, la cryptanalyse est l'étude des techniques qui permettent à une personne non autorisée de **décrypter** (ce mot s'emploie uniquement dans ce contexte) des messages chiffrés.

La **cryptographie** permet aussi de **signer** et d'**authentifier** des documents. En signant un document, on permet au destinaire de vérifier la provenance du document, et l'authentification permet de vérifier que ce document n'a pas été altéré.

Vous prendrez garde à ne pas confondre chiffrement et **codage**. Coder revient à remplacer chaque symbole (ou suite de symboles) par un(e) autre. Cela peut servir en cryptographie, mais la réciproque est fausse.

Un **chiffre** est la donnée d'un algorithme de chiffrement (et/ou de signature) et de l'algorithme de déchiffrement (et/ou de vérification et d'authentification). On utilise aussi le mot **protocole**, ou encore **algorithme**.

Nous appellerons **tiers non autorisé** toute personne à qui le message chiffré n'est pas destiné et qui pourrait, pour des raisons qui lui appartiennent, être tentée de le lire.

Un **canal non sécurisé** est un moyen de transmission de l'information dont la sûreté n'est pas garantie, c'est-à-dire sur lequel il est aisé pour un tiers non autorisés d'intercepter les messages. Les protocole de chiffrement trouve tout leur intérêt lorsque l'on doit faire transiter par un canal non sécurisé des informations confidentielles.

Un message, avant le chiffrement est appelé **message clair** ou encore **message en clair**. Une fois chiffré, on l'appelle **message chiffré**. Lorsqu'il a été déchiffré (a priori par son destinataire), on l'appelle **message déchiffré**.

## 11.2 La stéganographie

La **stéganographie** est l'art de dissimuler des messages, par exemple en utilisant les bits de poids faible d'une image au format `bmp` [9], ou encore en plaçant des informations dans le slack [10]. La stéganographie n'est pas considéré comme de la cryptographie, car le **message doit pouvoir tomber entre les mains d'un tiers non autorisée sans que celui-ci puisse le lire**. Ce critère n'est évidemment pas satisfait par la stéganographie, bien que celle-ci ait eu son heure de gloire pendant l'antiquité.

## 11.3 Les chiffres symétriques sans clé

Un chiffre symétrique sans clé se compose d'une fonction $f$ qui a tout message clair $M$ associe un message $f(M)$. Le message se déchiffre avec une fonction $f^{-1}$. Par exemple, le chiffre de César,qui consiste à remplacer toute lettre par celle qui se trouve trois positions plus loin dans l'alphabet. Cette méthode résistera aux attaques un certain temps, mais une fois la fonction $f$ découverte, il est trivial de déterminer $f^{-1}$, et toute la méthode est à jeter à la poubelle. Un

deuxième critère devant être satisfait est que **même si des personnes non autorisées connaissent l'algorithme chiffrement, elle ne doivent pas être capable de déchiffrer un message chiffré**.

**Exercice 1 - Chiffrement par transposition**

Chiffrer un message par transposition revient appliquer un algorithme de permutation des lettres.

1. Est-il aisé de cryptanalyser un message chiffré de la sorte ?
2. Considérez-vous qu'il s'agit d'un moyen de chiffrement sûr ?

**Exercice 2 - La barrière de la langue**

Pendant la deuxième guerre mondiale, les américains ont utilisé pour communiquer par radio la langue des indiens Navajos, très complexe et ne ressemblant à aucune autre pour communiquer rapidement et sûrement sans que les japonais ne puisse comprendre les conversations. Qu'en pensez-vous ?

## 11.4 Les chiffres symétriques à clé

On cherche une méthode de chiffrement telle que même si le secret de l'algorithme n'est pas suffisament bien tenu, cela n'aura pas de conséquences sur sa fiabilité. Un algorithme symétrique à clé est une fonction $f$ qui à tout message clair $M$ et à toute clé $K$ associe un message chiffré $C = f(M, K)$. On déchiffre le message en utilisant la même clé (ou une clé facile à trouver à partir de la première), $f^{-1}(C, K)$ nous donne le message déchiffré. Cela signifie que même en connaissant $f$, un tiers non autorisé ne pourra pas déchiffrer $C$ sans $K$. Si $K$ est découverte, il suffit de changer de clé, et il sera alors possible d'utiliser le même algorithme.

### 11.4.1 Le chiffre de Vigenère

La méthode de Vigenère est la plus connue.
– Un message $M$ est composé de nombres représentant des rangs de lettres dans l'alphabet
– Une clé $K$ est composée de $n$ nombres $k_1 k_2 \dots k_n$.
– On découpe $M$ en blocs de $n$ nombres $m_1 m_2 \dots m_n$
– A chaque bloc, on associe un bloc chiffré $c_1 c_2 \dots c_n$ avec $c_i = (m_i + k_i) mod 26$

Si $n = 10$, alors il existe $26^{10}$ possibilités, ce qui permet de bannir d'entré de jeu toutes les méthodes basées sur une énumération de toutes les clés. Soit $p$ le nombre de lettres chiffrées avec la clé $K$ que vous avez à votre disposition, mêmes si ces lettres proviennent de messages différents. Si le ratio $\frac{p}{n}$, est élevé, alors vous pouvez tenter de cryptanalyser les messages avec des méthodes statistiques [11].

### 11.4.2 Enigma

Enigma est une machine à chiffrer utilisée par l'armée allemande pendant la deuxième guerre mondiale. Elle se présentait comme une machine à écrire, mais écrivait d'autres lettres que celles qui étaient saisies. Un système de rotors qui pivotaient à chaque pression de touche changeait la clé au fur et à mesure que l'on avançait dans l'écriture du message. Cela ressemblait à un code de Vigenère dont la clé est aussi longue que le message, et comme les allemands changeaient de clé chaque semaine, cela rendait irréaliste toute cryptanalyse basée sur une analyse des fréquences. La clé utilisée était une disposition des rotors et une position initiale, ce qui faisait que même si une machine enigma (l'algorithme) tombait entre les mains d'un ennemi de l'armée allemande, celui-ci ne pouvait déchiffrer les messages sans connaître la disposition des rotors (la clé). Enigma a donné beaucoup de fil à retordre aux alliés, et était considéré comme un moyen de chiffrement très efficace.

## 11.5 Les algorithmes asymétriques

Mais quelque soit la fiabilité théorique d'algorithme symétrique à clé, un gros problème pratique se pose : que faire si les clés tombent entre les mains d'un tiers non autorisé ?

### 11.5.1 Un peu d'histoire[1]

Les allemands étaient obligés de faire circuler les clés en centaines d'exemplaires dans toute l'europe, en garantir la sécurité était une tâche très difficile. Et ce qui devait arriver arriva : des anglais s'emparèrent de clés d'enigma sans que les allemands le sachent. Comme ils disposaient de copies d'enigma depuis longtemps déjà et que les allemands ne se savaient pas écoutés, ils purent tranquillement déchiffrer tous les messages allemands. Nombreux sont ceux qui considèrent que cela a été décisif pour la victoire des alliés.

### 11.5.2 Le principe

Mettre des clés en circulation et garantir leur sécurité est considéré aujourd'hui comme un tâche impossible. Comment échanger des clés sans prendre le risque qu'un message soit décrypé par un tiers non autorisé ? L'idée de la réponse est d'utiliser deux clés : $K_e$ pour chiffrer et $K_d$ pour déchiffrer. Le destinataire du message est la seule personne à détenir $K_d$, elle diffuse $K_e$ librement. Un message $M$ est chiffré avec une fonction $f$ en un message $C = f(M, K_e)$. Son destinataire déchiffre le message avec une fonction $g$ telle que $M = g(C, K_d)$. Les deux fonctions $f$, $g$, et la clé $K_e$ doivent pouvoir être publiées et tomber entre les mains d'un tiers non autorisé sans qu'il puisse déchiffrer $M$. Autrement dit, il doit être difficile de trouver $K_d$ à partir de $f$, $g$, $K_e$ et $M$.

On peut se représenter ce type de protocole de la façon suivante. Alice veut envoyer un message à Bob. Bob lui envoie une boîte et un cadenas ouvert. Alice place le message dans la boîte et verrouille le cadenas. Maintenant, seul le détenteur de la clé du cadenas peut ouvrir la boîte. S'il s'agit de Bob, lui seul pourra récupérer le message, même Alice ne pourra pas le faire. Le principe le la cryptographie asymétrique est le même :

- $K_e$ est la **clé publique** et permet à toute personne de chiffrer un message.
- Seule la **clé privée** $K_d$ permet le déchiffrement du message.

#### Exercice 3 - SSH

Les chiffres à clé publique sont très gourmands en calculs, ils rendent inenvisageable le transfert de volumes importants de données. Par contre les protocoles symétriques sont très rapides en pratique. Comment conjuguer les deux pour communiquer de façon rapide et sécurisée à travers un canal non sécurisé.

# Chapitre 12

# GnuPG

## 12.1 Présentation

Gnu Privacy Guard[12] est un logiciel de chiffrement asymétrique et de gestion de clés. L'algorithme utilisé pour signer et chiffrer est El Gamal, qui est d'une fiabilité comparable à celle d'RSA.

Il s'installe très simplement avec `urpmi`. Il existe sous forme de plugin pour Thunderbird, et est donc muni d'une interface graphique. Nous l'utiliserons dans ce cours en ligne de commande, rassurez-vous il y en a très peu à apprendre.

GNU Privacy Guard est, comme son nom l'indique, sous licence GNU, vous pouvez donc en faire l'usage que vous voulez - y compris commercial - sans risquer de vous retrouver un jour au tribunal.

## 12.2 Utilisation

### 12.2.1 Génération d'une clé privée

Avant toute utilisation du GPG, vous devez créer une clé privée. GPG vous la générera et la conservera en lieu sûr... Pour générer votre clé privée, saisissez la commande :

```
gpg --gen-key
```

Plusieurs informations vous seront demandées, y compris un mot de passe pour déverouiller la clé privée, il est impératif que ce mot de passe soit robuste, sinon un tiers non autorisé ayant un accès physique à votre machine aurait accès à votre clé privée. Cette clé est stockée dans un répertoire de votre `home`, elle sera utilisé à chaque déchiffrement et à chaque signature.

### 12.2.2 Exportation de la clé publique

Si vous souhaitez recevoir des messages chiffrés,vous devez donner votre clé publique à vos correspondants. Pour exporter votre clé publique dans un fichier, saisissez la commande

```
gpg --export --armor --output nomdufichier
```

L'option `--armor` encode la clé dans un format affichable et pouvant être envoyé par mail. Vous pouvez donner ce ficher à qui vous voulez et même le mettre sur votre page web, cette clé permet seulement de chiffrer des fichiers, seule votre clé privée permettra leur déchiffrement.

### 12.2.3 Importation

Si vous souhaitez envoyer des fichiers chiffrée à un correspondant, vous aurez besoin de sa clé publique. Il lui suffit de vous l'envoyer d'une façon similaire à celle dont vous lui aurez fourni votre clé publique. Une fois sa clé publique entre vos mains, il ne vous reste plus qu'à saisir

```
gpg --import --armor nomdufichier
```

### 12.2.4 Liste des clés

La liste des clés publiques en votre possession s'obtient en saisissant

```
gpg --list-keys
```

### 12.2.5 Chiffrement

Une fois que vous detenez la clé publique d'un de vos correspondants, vous avez la possibilité de chiffrer des fichiers que lui seul pourra déchiffrer.

```
gpg --encrypt --recipient destinataire --output fichierChiffré fichieràChiffrer
```

`destinataire` est une sous-chaine permettant d'identifier la clé publique du destinataire du message chiffré, `fichierChiffré` est le nom du fichier généré par cette commande, `fichieràChiffrer` est, comme son nom l'indique, le nom du fichier à chiffrer.

### 12.2.6 Déchiffrement

Vous pouvez déchiffrer un fichier chiffré avec votre clé publique en utilisant la commande

```
gpg --decrypt --output fichierDéchiffré fichierChiffré
```

### 12.2.7 Signature

Les messages sont automatiquement signés avec votre clé privée lors du chiffrement. La signature est vérifiée lors du déchiffrement, à condition bien sûr, que le destinataire ait en sa possession la clé publique de l'émetteur.

### 12.2.8 Bref

Ces notes n'étaient qu'un survol de GnuPG, pour plus d'informations, ou en vue d'une utilisation professionnelle, se référer à [13].

#### Exercice 1 - Création des clés

1. Installez GPG avec urpmi si vous êtes sous linux, et débrouillez-vous si vous êtes sous windaube.
2. Créez un couple de clés.

#### Exercice 2 - Echange de clés

1. Trouvez un binôme
2. Exportez vos clés publiques et échangez-les par mail.
3. Documentez-vous sur l'option `fingerprint`, expliquez quelle utilisation vous pourriez bien en faire.

#### Exercice 3 - Echange de messages

1. Placez dans des fichiers texte des messages arbitraires
2. Chiffrez-les et échangez-les.
3. Déchiffrez les messages.

**Exercice 4 - Elargissement de la toile de confiance**

Je désignerai dorénavant votre binôme par la lettre $B$.

1. Trouvez un trinôme $C$ ($B$ devra aussi trouver un trinôme $D$).
2. Récupérez la clé publique $P_C$ de $C$, vous prendrez soin de vérifier son authenticité avec son `fingerprint`
3. Chiffrez la clé publique $P_B$ de $B$ et envoyez-la à $C$.
4. $C$ devra déchiffrer $P_B$ et envoyer sa clé publique chiffrée à $B$.
5. $B$ devra déchiffrer la clé publique de $C$ et envoyer un message chiffré de confirmation à vous et à $C$.

# Chapitre 13

# GMP

## 13.1 Présentation

GMP[14] est un package d'arithmétique multi-précision, c'est à dire permettant de représenter des entiers, des rationnels et des flottants avec une précision arbitraire, et non pas celle préconisée par les normes de représentation des entiers et des flottants. Les algorithmes de cryptographie manipulent des nombres de plusieurs dizaines, voire centaines de chiffres, ce qui nécessite l'utilisation d'une telle bibliothèque.

Vous pouvez l'installer très simplement avec `urpmi` ou `apt-get`. Comme ce package est sous licence GNU, vous pouvez en faire l'utilisation que vous voulez. La documentation est disponible en anglais et vous donne la liste détaillée des fonctions. Le but de ce document est juste de vous faire comprendre le fonctionnement du package en quelques minutes.

## 13.2 Quelques points délicats

Le point le plus complexe est la gestion de la mémoire, vous devez, en manipulant les fonctions de GMP.h, être très attentifs aux valeurs que vous passez en paramètre.

### 13.2.1 Compilation

N'ouliez pas d'importer `gmp.h` et d'ajouter l'option de compilation `-lgmp`. Sinon ça marche pas.

### 13.2.2 Initialisation

Le type `mpz_t` vous est fourni avec la librairie, méfiez-vous de l'absence d'étoile, il y a nécessairement des pointeurs derrière ce type. Des fonctions préfixées par `mpz_init` vous permettent d'initialiser ces variables. N'oubliez jamais l'initialisation, elle correspond à l'allocation dynamique, elle n'oubliez jamais la destruction, qui elle correspond à la libération de la mémoire.

```
mpz_t i; // declaration
mpz_init(i); // allocation
```

Les instructions ci-dessus déclarent un entier multiprécision $i$ et effectuent l'allocation dynamique. La fonction `mpz_init` initialise $i$ à 0. Ne pas confondre initialisation et affectation. Un fois la variable initialisée, vous pouvez lui affecter toutes les valeurs que vous voulez. Par contre n'initialisez pas une variable déjà initialisée, vous devez préalablement la détruire. On détruit un entier multiprécision (bref, on libère la mémoire) avec la fonction `mpz_clear`.

```
mpz_t i; // declaration
mpz_init(i); // allocation
mpz_set_ui(i, 5000); // affectation
/*
        Utilisation de i
```

```
*/
mpz_clear(i); // liberation de la memoire
```

### 13.2.3 Les fonctions

Vous remarquerez dans la documentation que presque toutes les fonctions de manipulation d'entiers multiprécision retounent `void`. Il est conseillé avec ce type de variable de ne pas utiliser les valeurs de retour, mais d'ajouter un paramètre pour y placer le résultat. Par exemple, pour multiplier un entier multiprécision par un `long`, on procède de la sorte :

```
// declarations
mpz_t resultat;
mpz_t operandeGauche;
long operandeDroite;

// allocations
mpz_init(resultat);
mpz_init(operandeGauche);

// affectations
mpz_set_ui(operandeGauche, 1000);
operandeDroite = 100;

// calcul
mpz_mul_si(resultat, operandeGauche, operandeDroite);
```

Les instructions ci-dessus effectue le calcul $resultat = operandeGauche \times operandeDroite$, soit $resultat = 1000 \times 100$. Vous remarquez le résultat de la multiplication est le premier paramètre, et non pas la valeur de retour.

### 13.2.4 Affichage

Pour afficher ou ecrire dans un fichier la valeur d'un entier multiprécision, on utilise la fonction `mpz_out_str (FILE *stream, int base, mpz t op )`. Si `stream` est égal à `NULL`, la valeur est ecrite dans `stdout`.

```
mpz_t valeur;
mpz_init(valeur);
mpz_set_ui(valeur, 17);
mpz_out_str(NULL, 10, valeur);
mpz_clear(valeur);
```

Les instructions ci-dessus affichent 17.

### 13.2.5 Exemple

Voici un petit exemple avec quelques calculs sur des entiers multi-précision.

```
#include<gmp.h>
#include<stdio.h>

/*
  Place la factorielle de n dans res. res doit etre initialise
 */

void factorielle(long n, mpz_t res)
{
  mpz_t temp;
  if (n == 0)
```

```
    {
      mpz_set_ui(res, 1);
    }
  else
    {
      mpz_init(temp);
      factorielle(n-1, temp);
      mpz_mul_si(res, temp , n);
      mpz_clear(temp);
    }
}

/*
  Place b^n dans res. res doit etre initialise
 */

void puissance(long b, long n, mpz_t res)
{
  mpz_t temp;
  if (n == 0)
    {
      mpz_set_ui(res, 1);
    }
  else
    {
      mpz_init(temp);
      puissance(b, n-1, temp);
      mpz_mul_ui(res, temp, b);
      mpz_clear(temp);
    }
}

/*
  Affiche les 100 premieres puissances de 10.
*/

int main()
{
  mpz_t i;
  long ind;
  mpz_init(i);
  for(ind = 0 ; ind <= 100 ; ind++)
    {
      puissance(10, ind, i);
      mpz_out_str(NULL, 10, i);
      printf("\n");
    }
  mpz_clear(i);
  return 0;
}
```

Et voici, pour vous éviter de vous prendre la tête, le makefile.

```
all : testGMP.c
        gcc -g -lgmp testGMP.c
```

```
run: all
	./a.out
```

### 13.2.6 Pour terminer

Etant donné le nombre de sous-programmes disponibles dans GMP, il serait inutile de les copier/coller ici. Je vous laisse consulter la documentation [15]. Bon courage.

**Exercice 1 - Prise en main**

Ecrivez un programme qui affiche la table de multiplication de $10^{20}$

**Exercice 2 - Fonction puissance**

Reprennez la fonction puissance dans l'annexe sur GMP et modifiez-là en utilisant le fait que si $n$ est pair alors,

$$b^n = (b^2)^{\frac{n}{2}}$$

Modifiez la fonction puissance de sorte qu'elle prenne des entiers multiprécision en paramètre.

**Exercice 3 - La fonction d'Ackermann**

La fonction d'Ackermann $\phi$, est définie comme suit :
- $\phi(0, n) = n + 1$
- $\phi(m, 0) = \phi(m - 1, 1)$
- $\phi(m, n) = \phi(m - 1, \phi(m, n - 1))$

Utilisez GMP pour calculer le plus de valeurs possibles de la suite d'Ackermann.

# Chapitre 14

# Arithmétique

Les bases mathématiques permettant la compréhension des algorithmes asymétriques sont exposées dans ce cours. Vous trouverez un exposé plus détaillé dans [2]. Notez aussi que ce cours se recoupe avec le programme de spé maths de terminale S.

Sauf mention explicité du contraire, tous les nombres considérés ici sont des entiers relatifs, c'est-à-dire des nombres entiers pouvant être positifs, négatifs ou nuls.

## 14.1 Divisibilité

**Définition 14.1.1** *$a$ **divise** $b$, s'il existe $k$ tel que $ak = b$*

$a$ divise $b$ se note $a|b$. On dit alors que $a$ est un diviseur de $b$, et que $b$ est un multiple de $a$.

**Exercice 1**

Prouvez, en utilisant la définition de la divisibilité, que
- $3|21$
- $(-3)|6$

**Exercice 2**

Prouvez les propriétés suivantes :

1. Si $a|b$ et $b|a$, alors $|a| = |b|$
2. Si $a|b$ et $b|c$, alors $a|c$
3. Si $a|b$ et $a|c$, alors $a|(b + c)$
4. Si $a|b$ et $a|c$, alors $a|bc$
5. Si $a|b$, alors $a^k|b^k$ pour tout entier naturel $k$.

## 14.2 Division euclidienne

**Définition 14.2.1** *Soient $a$ un nombre relatif et $b$ un entier strictement positif, on effectue la **division** $a$ par $b$ en déterminant $q$ et $r$ tels que $a = qb + r$, avec $0 \leq r < q$. $q$ est le **quotient**, $r$ le **reste**.*

**Exercice 3**

Effectuez les divisions suivantes :
- 12 par 5
- 11 par 2
- 13 par 13
- 2 par 4

– $(-6)$ par 2
– 0 par 3
– $(-5)$ par 2

## 14.3 PGCD

**Définition 14.3.1** *Le plus grand commun diviseur de $a$ et $b$, noté $pgcd(a, b)$ est le plus grand nombre qui divise à la fois $a$ et $b$.*

**Exercice 4**

Complétez le tableau suivant :

| $a$ | $b$ | $pgcd(a, b)$ |
|---|---|---|
| 6 | 3 | |
| 3 | 6 | |
| 17 | 13 | |
| $2^9$ | $2^6$ | |
| 0 | 4 | |
| 7 | 7 | |

## 14.4 Nombres premiers

**Définition 14.4.1** *Un nombre $p \geq 2$ est premier s'il n'a que par 1 et $p$ comme diviseurs positifs.*

Par convention, 1 n'est pas premier.

**Exercice 5**

Les nombres suivants sont-ils premiers ?
– 4
– 7
– 23
– 1

**Exercice 6**

Prouvez qu'il existe une infinité de nombres premiers.

## 14.5 Nombres premiers entre eux

**Définition 14.5.1** *$a$ et $b$ sont premiers entre eux si $pgcd(a, b) = 1$*

Autrement dit, $a$ et $b$ sont premiers entre eux s'ils n'ont aucun autre diviseur commun que 1.

**Exercice 7**

Les couples de nombres suivants sont-ils premiers entre eux ?
– 3 et 5
– 20 et 32
– 23 et 41
– 18 et 27
– 0 et 4
– 7 et 7
Existe-t-il des nombres premiers avec eux-mêmes ?

**Exercice 8**

Prouvez que deux nombres premiers distincts sont premiers entre eux.

**Exercice 9**

Soient $a$ et $b$ deux entiers relatifs, $p = pgcd(a, b)$, $a' = \frac{a}{p}$ et $b' = \frac{b}{p}$. Prouvez que $a'$ et $b'$ sont premiers entre eux.

## 14.6 Décomposition en produit de facteurs premiers

**Propriété 14.6.1** *Tout nombre entier positif $x$ peut s'écrire comme un produit de nombres premiers. C'est-à-dire sous la forme*

$$x = p_1^{e_1} p_2^{e_2} \dots p_i^{e_i} \dots p_n^{e_n}$$

*où pour tout $i \in \{1, \dots, n\}$, $p_i$ est le $i$-ème nombre premier et $e_i$ un nombre positif ou nul.*

Tout nombre est défini de façon unique par la suite $(e_1, e_2, \dots, e_n, 0, 0, \dots)$. Par exemple, quel nombre est défini par la suite $(1, 3, 0, 1, 2, 0, 0, \dots)$ ?

**Exercice 10**

Décomposez chacun de ces nombres en produit de facteurs premiers.
- 21
- 286
- 51
- 1344

**Exercice 11**

Donnez les nombres correspondant aux décompositions en facteurs premiers suivantes.
- $(0, 0, 1, 1, 0, \dots, 0, \dots)$
- $(1, 1, 2, 0, \dots, 0, \dots)$
- $(1, 0, 2, 0, 1, 0, \dots, 0, \dots)$
- $(0, 0, 1, 0, \dots, 0, \dots)$
- $(0, \dots, 0, \dots)$

### 14.6.1 Application à la divisibilité

**Propriété 14.6.2** *Soient*

$$x = p_1^{e_1} p_2^{e_2} \dots p_i^{e_i} \dots p_n^{e_n}$$

*et*

$$y = p_1^{f_1} p_2^{f_2} \dots p_i^{f_i} \dots p_n^{f_n}$$

*$x|y$ si et seulement si pour tout $i$, $e_i \leq f_i$*

**Exercice 12**

Donnez la liste des diviseurs du nombre correspondant à la décomposition en facteurs premiers suivante :

$$(1, 0, 1, 0, 1, 0, \dots, 0, \dots)$$

**Exercice 13**

Soient $x = p_1^{e_1} p_2^{e_2} \dots p_i^{e_i} \dots p_n^{e_n}$ et $y = p_1^{f_1} p_2^{f_2} \dots p_i^{f_i} \dots p_n^{f_n}$. Donnez la décomposition en facteurs premiers de $xy$.

### 14.6.2 Application au calcul du PGCD

**Propriété 14.6.3** *Soient*

$$a = p_1^{e_1} p_2^{e_2} \dots p_i^{e_i} \dots p_n^{e_n}$$

*et*

$$b = p_1^{f_1} p_2^{f_2} \dots p_i^{f_i} \dots p_n^{f_n}$$

*alors*

$$pgcd(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \dots p_i^{\min(e_i,f_i)} \dots p_n^{\min(e_n,f_n)}$$

**Exercice 14**

Calculez les PGCDs des couples de nombres dont les décompositions en facteurs premiers sont données :
- $(0,0,2,1,0,\dots,0,\dots)$ et $(0,0,1,2,0,\dots,0,\dots)$
- $(1,1,1,0,\dots,0,\dots)$ et $(0,2,2,2,0,\dots,0,\dots)$
- $(1,0,1,0,\dots,0,\dots)$ et $(0,1,0,1,0,\dots,0,\dots)$
- $(2,2,1,1,0,\dots,0,\dots)$ et $(0,0,1,1,0,\dots,0,\dots)$

**Exercice 15**

Soient $x = p_1^{e_1} p_2^{e_2} \dots p_i^{e_i} \dots p_n^{e_n}$ et $y = p_1^{f_1} p_2^{f_2} \dots p_i^{f_i} \dots p_n^{f_n}$. Déterminer une condition nécessaire et suffisante sur les suites $e$ et $f$ permettant de savoir si $x$ et $y$ sont premiers entre eux.

## 14.7 Théorème de Bezout

**Théorème 14.7.1** *Pour tous a, b relatifs, il existe u et v relatifs tels que*

$$au + bv = pgcd(a,b)$$

**Théorème 14.7.2** *a et b sont premiers entre eux si et seulement s'il existe u et v relatifs tels que*

$$au + bv = 1$$

### 14.7.1 Théorème de Gauss

**Théorème 14.7.3** *Si a et b sont premiers entre eux, et $a|bc$, alors $a|c$*

## 14.8 Algorithme d'Euclide

### 14.8.1 Algorithme d'Euclide pour le PGCD

**Propriété 14.8.1** *Pour tous a et b relatifs,*
- $pgcd(a,0) = a$
- $pgcd(a,b) = pgcd(b, a \bmod b)$

**Exercice 16**

Appliquez l'algorithme d'Euclide aux couples de nombres suivants :
- 3 et 6
- 4 et 0
- 161 et 184
- 21 et 34

**Exercice 17**

On admet que le $pgcd(a, b)$ est la plus petite combinaison linéaire strictement positive de $a$ et $b$.

1. Montrer que $pgcd(a, b)|pgcd(b, a \ mod \ b)$
2. Montrer que $pgcd(b, a \ mod \ b)|pgcd(a, b)$
3. Montrer que l'algorithme d'Euclide se termine.
4. Soit $F_n$ la suite définie par
   - $F_0 = 0$
   - $F_1 = 1$
   - $F_n = F_{n-1} + F_{n-2}$, si $n \geq 2$

   On appelle $F_n$ le $n$-ième nombre de Fibonacci. Montrez par récurrence que
   $$F_n = \frac{1}{\sqrt{5}}(\phi^n + \psi^n)$$
   avec
   $$\phi = \frac{1+\sqrt{5}}{2}; \psi = \frac{1-\sqrt{5}}{2};$$
   $F$ est-elle croissante ?
5. Prouvez que pout tout $n \geq 2$,
   $$F_n \ mod \ F_{n-1} = F_{n-2}$$
6. Prouvez par récurrence que
   $$pgcd(F_n, F_{n-1}) = 1$$
7. Prouvez par récurrence que le calcul de
   $$pgcd(F_n, F_{n-1})$$
   nécessite $n$ appels récursifs.
8. Prouvez par récurrence que si $F_n \geq k$ et $F_{n-1} \geq k'$, alors
   $$pgcd(k, k')$$
   se calcule avec un nombre d'appels récursifs inférieur ou égal à $n$.
9. Prouvez que si $k > k'$, alors $pgcd(k, k')$ s'exécute en $\mathcal{O}(log_\phi(k))$
10. Montrez que l'hypothèse $k > k'$ n'était pas restrictive.

### 14.8.2 Algorithme d'Euclide étendu

Il possible d'enrichir l'algorithme d'Euclide afin qu'il calcule les coefficients $u$ et $b$ de la relation de Bezout. Étant donnés deux nombres $a$ et $b$,

- Si $b = 0$, alors $pgcd(a, b) = a$, et on a $1.a + 0.b = pgcd(a, b)$
- Par récurrence, appliquons la même méthode au couple $(b, a \ mod \ b)$, alors on a $u$ et $v$ tels que
  $$u.b + v.(a \ mod \ b) = pgcd(b, a \ mod \ b)$$
  Soit $\lfloor \frac{a}{b} \rfloor$ le quotient de la division de $a$ par $b$, alors
  $$a \ mod \ b = a - \lfloor \frac{a}{b} \rfloor b$$
  Donc
  $$u.b + v.(a - \lfloor \frac{a}{b} \rfloor b) = pgcd(b, a \ mod \ b)$$
  Ce qui se réécrit
  $$v.a + (u - v\lfloor \frac{a}{b} \rfloor)b = pgcd(b, a \ mod \ b)$$

Comme, de plus,

$$pgcd(a, b) = pgcd(b, a \ mod \ b)$$

alors

$$v.a + (u - v\lfloor\frac{a}{b}\rfloor)b = pgcd(a, b)$$

Donc, en posant $u' = v$ et $v' = u - v\lfloor\frac{a}{b}\rfloor$, on a bien

$$u'.a + v'.b = pgcd(a, b)$$

**Exercice 18**

Appliquez l'algorithme d'Euclide étendu aux couples de nombres suivants. Lesquels sont premiers entre eux ?
- 3 et 6
- 4 et 0
- 161 et 184
- 21 et 34

**Exercice 19**

Ecrivez en C l'algorithme d'Euclide étendu (passez $u$ et $v$ en paramètre par référence) sans GMP.

**Exercice 20**

Prouvez le théorème de Gauss

### 14.8.3 Morceaux choisis

**Exercice 21 - Divisbilité**

Prouvez que si $a|b$ alors soit $|a| \leq |b|$, soit $b = 0$.

**Exercice 22 - Unicité de la division**

Démontrer que le couple $(q, r)$ résultant de la division de $a$ et $b$ est unique.

**Exercice 23 - Divisibilité par** 10

Combien y a-t-il de 0 à la fin de (1000!) ?

**Exercice 24 - Combien y a-t-il de nombres premiers ?**

1. Soit $u_n = p_1 p_2 \dots p_n - 1$ le produit des $n > 1$ plus petits nombres premiers auquel on a retranché 1, $p_i$ un des $n$ plus petits nombres premiers. Effectuez une division eulidienne de $u_n$ par $p_i$.
2. Prouvez que $u_n$ n'est divisible par aucun des $n$ plus petits nombres premiers.
3. En déduire qu'il existe une infinité de nombres premiers.

### 14.8.4 Applications avec GMP

Dans tous les exercices qui suivent, vous n'aurez droit qu'au type `mpz_t`.

**Exercice 25 - Algorithme d'Euclide étendu**

Programmez l'algorithme d'Euclide étendu en affichant une trace d'exécution montrant les étapes du calcul de $u$ et $v$.

**Exercice 26 - Expérimentation avec la suite de Fibonacci**

Ecrivez une fonction retournant le $n$-ème nombre de Fibonacci en utilisant la relation de récurrence donnée dans la section "algorithme d'Euclide". Que remarquez-vous lorsque l'on calcule des valeurs de la suite de Fibonacci avec de grandes valeurs de $n$ (control-C pour interrompre l'exécution d'un programme :-) ? On considère, pour remédier à ce petit problème la fonction $\phi$ définie comme suit

- $\phi(a, b, 0) = b$
- $\phi(a, b, n) = \phi(a + b, a, n - 1)$

Prouvez par récurrence sur $n$ que $F_{n+k} = \phi(F_{k+1}, F_k, n)$, puis que $F_n = \phi(1, 0, n)$. Utilisez cette relation pour modifier la fonction calculant le $n$-ème nombre de Fibonacci. Tester l'algorithme d'Euclide étendu avec des couples $(F_n, F_{n-1})$ (prenez de grandes valeurs de $n$).

# Chapitre 15

# Arithmétique modulaire

Les protocoles de chiffrement sont basés sur des opérations dans des espaces quotient, détaillés dans la dernière partie de ce cours. Les définitions sont simplifiées au minimum nécessaire pour comprendre ce cours, et donc moins précises que ce que vous pourriez trouver dans des ouvrages spécialisés.

La première section, sur les congruences, expose des points du programme de spé maths de terminale S, rédigés toutefois de façon plus technique. Les deux sections suivantes abordent des éléments du programme de première année de fac, ou de maths sup, seuls les points essentiels à la compréhension de la dernière section du cours sont exposées, j'ai pris la liberté de simplifier quelques définitions. La dernière section expose des méthodes utilisées directement en cryptographie asymétrique, elles sont aussi enseignées en première année de fac.

## 15.1 Congruences

**Définition 15.1.1**

$$a \equiv b \pmod{n}$$

*si*

$$n|(a-b)$$

*On dit alors que $a$ est **congru** à $b$ modulo $n$.*

Autrement dit, $a$ est $b$ congru à $b$ modulo $n$ si et seulement si $a$ et $b$ ont le même reste de la division par $n$. Plus formellement,

**Propriété 15.1.1** *$a \equiv b \pmod{n}$ si et seulement si $a \bmod n = b \bmod n$.*

En effet, si $n|(a-b)$, alors il existe $k$ tel que $kn = a - b$. Divisons $a$ et $b$ par $n$, il existe $q$, $q'$, $r$ et $r'$ tels que $a = qn + r$ et $b = q'n + r'$, avec $0 \leq r < n$ et $0 \leq r' < n$. Donc $kn = a - b$ équivaut à

$$kn = qn + r - (q'n + r')$$

ssi

$$kn = (q - q')n + (r - r')$$

ssi

$$n(k + q' - q) = (r - r')$$

Donc

$$n|(r - r')$$

Or, $|r - r'| < n$, donc $n$ ne peut diviser $(r - r')$ que si $r - r' = 0$. On a donc $r = r'$, comme $r = a \bmod n$ et $r' = b \bmod n$, alors $a \bmod n = b \bmod n$.

Réciproquement, supposons $a \bmod n = b \bmod n$. Divisons $a$ et $b$ par $n$, il existe $q$, $q'$ et $r$ tels que $a = qn + r$ et $b = q'n + r$, avec $0 \leq r < n$. Donc

$$a - b = qn + r - (q'n + r)$$

ssi

$$a - b = qn - q'n$$

ssi

$$a - b = (q - q')n$$

Donc, $n|(a - b)$.

**Propriété 15.1.2** *Si* $a \equiv b \pmod{n}$ *et* $a' \equiv b' \pmod{n}$*, alors*

1. $a + a' \equiv b + b' \pmod{n}$
2. $aa' \equiv bb' \pmod{n}$
3. $a^k \equiv b^k \pmod{n}$

**Exercice 1**

Pour chacune de ces propriétés,
– donnez un exemple numérique
– donnez une preuve

**Exercice 2**

Calculez les valeurs suivantes :

$$73^3 \ mod\ 10; 4.7 \ mod\ 9; 14 + 71 \ mod\ 83; 4^4 \ mod\ 12;$$

$$3.9 - 7.8 \ mod\ 11; 247^{349} \ mod\ 7; 2^{1247} \ mod\ 17;$$

$$34^{57} - 1 \ mod\ 11; 9518^{42} - 4 \ mod\ 5$$

$$3^{2n+1} + 5^{2n+1} \ mod\ 4; 6^n + 13^{n+1} \ mod\ 7$$

## 15.2 Rappels d'algèbre

### 15.2.1 Monoïdes

**Définition 15.2.1** *Un monoïde* $(E, +, 0)$ *est un ensemble* $E$ *muni d'une addition* $+$ *et d'un élément neutre* $0$ *pour cette addition.*

Un élément neutre 0 est tel que $\forall e \in E, e + 0 = 0 + e = e$.

Par exemple, l'ensemble des entiers naturels, noté $N$ (ou $(N, +, 0)$), est un monoïde. En effet,
– il est toujours possible d'additionner deux entiers naturels
– et 0 est un élément neutre pour l'addition.
L'ensemble $(B, \times, 1)$ des booléens, muni du produit est aussi un monoïde. Il est en effet toujours possible de multiplier deux booléens, et on a $0.1 = 0$, $1.1 = 1$, donc 1 (vrai) est un élément neutre pour le produit booléen.

### 15.2.2 Groupes

**Définition 15.2.2** $(G, +, 0)$ *est un groupe si*
– $(G, +, 0)$ *est un monoïde*
– *pour tout élément* $x$ *de* $G$*, il existe un élément* $-x$ *appelé "**opposé** de* $x$*", tel que*

$$x + (-x) = 0$$

Autrement dit, un groupe est un monoïde dans lequel il existe une soustraction. On remarque que $N$ n'est pas un groupe mais que l'ensemble des entiers relatifs $Z$ (ou $(Z, +, 0)$) est bien un groupe. En effet,
– il est toujours possible d'additionner deux entiers relatifs
– 0 est bien un élément neutre pour l'addition.
– pour tout $x \in Z$, $-x$ existe, ce qui n'est pas le cas dans $N$.

### 15.2.3 Anneaux

**Définition 15.2.3** $(A, +, \times, 0, 1)$ *est un anneau si*
- $(A, +, 0)$ *est un groupe*
- *on peut toujours multiplier deux éléments de* $A$
- *il existe un élément neutre* $1$ *pour la multiplication*

Par exemple, l'ensemble des entiers relatifs est un anneau, il suffit de prendre 1 comme élément neutre pour la multiplication. L'ensemble des matrices carrées est un anneau.

### 15.2.4 Corps

**Définition 15.2.4** $(C, +, \times, 0, 1)$ *est un corps si*
- $(G, +, \times, 0, 1)$ *est un anneau*
- *pour tout élément* $x$ *de* $C$*, sauf* $0$*, il existe un élément* $x^{-1}$ *appelé "**inverse** de* $x$*", tel que*

$$x \times (x^{-1}) = 1$$

Autrement dit, $C$ est un corps s'il est muni d'une division. Par exemple, l'ensemble des entiers relatifs n'est pas un corps, mais l'ensemble des rationnels en est un. L'ensemble des matrices inversibles est un corps.

**Exercice 3**

On considère l'ensemble $(\mathcal{F}, +, \circ, f_0, f_1)$ des fonctions linéaires ($f(x) = ax + b$) muni de l'addition ($(f + g)(x) = f(x) + g(x)$) et de la composition $\circ$ ($(f \circ g)(x) = f(g(x))$).
- Est-ce que $(\mathcal{F}, +, \circ, f_0, f_1)$ est un monoïde ?
- un groupe ?
- un anneau ?
- un corps ?
- existe-t-il un sous-ensemble de $\mathcal{F}$ formant un corps ?

**Exercice 4**

On considère l'ensemble $(\mathcal{P}, +, \circ, f_0, f_1)$ des polynômes à coefficients réels

$$P(x) = \sum_{i=1}^{n} a_i x^i + a_0$$

muni de l'addition et de la composition
- Est-ce que $(\mathcal{P}, +, \circ, f_0, f_1)$ est un monoïde ?
- un groupe ?
- un anneau ?
- un corps ?
- existe-t-il un sous-ensemble de $\mathcal{P}$ formant un corps ?

### 15.2.5 Relations d'équivalence

**Définition 15.2.5** *Une relation* $\sim$ *sur* $E$ *est une **relation d'équivalence** si* $\sim$

1. $\sim$ *est **réflexive**, c'est à dire* $\forall e \in E,\ e \sim e$
2. $\sim$ *est **symétrique**, c'est à dire* $\forall e, e' \in E, e \sim e' \Rightarrow e' \sim e$
3. $\sim$ *est **transitive**, c'est à dire* $\forall e, e', e'', e \sim e'$ *et* $e' \sim e'' \Rightarrow e \sim e''$

Par exemple, la relation $=$ sur les réels est une relation d'équivalence. Il se trouve que la relation $\equiv$ sur les entiers relatifs est une relation d'équivalence, preuve en exercice.

**Exercice 5**

Etant donné un entier relatif non nul $n$. Pour tous $a$, $b$ entiers relatifs, on pose $a \sim b$ si $a \equiv b \pmod{n}$. Prouver que $\sim$ est une relation d'équivalence.

## 15.3 Espaces quotients

### 15.3.1 Classes d'équivalence

Une relation d'équivalence permet de répartir tous les éléments d'un ensemble dans des ensembles appelés **classes d'équivalence**. Deux éléments $a$ et $b$ sont dans la même classe si $a \sim b$. Par exemple, si $n = 5$, on peut répartir tous les relatifs dans 5 classes d'équivalence selon le reste de leur division par 5. Par exemple,

$$c = \{\ldots, -13, -8, -3, 2, 7, 12, 17, \ldots\}$$

est une classe d'équivalence, tous ses éléments sont, deux à deux, distants d'un multiple de 5. Pour désigner une classe d'équivalence, on utilise un élément de cette classe, appelé **représentant**. En prenant 2 comme représentant, on note alors $[2]_5$ la classe des équivalents à 2 modulo 5. On aurait pu aussi utiliser un autre élément de $c$ comme représentant, mais ce n'est pas usuel, on prend en général un représentant compris entre 0 et $n-1$.

**Définition 15.3.1** *Soit $n > 0$, et $k \in \{0, \ldots, n-1\}$, alors on note $[k]_n = \{e | e \sim k\}$.*

### 15.3.2 $Z/nZ$

Supposons qu'en calculant $c = a + b \ mod \ n$, on substitue à $a$ et à $b$ des éléments de même classe d'équivalence, alors le résultat est-il de la même classe que $c$ ? Vérifions : remplaçons $a$ par $a' \in [a]_n$ et $b$ par $b' \in [b]_n$. Comme $a \sim a'$ et $b \sim b'$, alors il existe $k$ et $k'$ tels que $a' = a + kn$ et $b' = b + k'n$. Calculons

$$\begin{aligned} c' &= a' + b' \\ &= a + kn + b + k'n \\ &= a + b + (k + k')n \\ &= c + (k + k')n \end{aligned}$$

Donc $c \sim c'$ et $c' \in [c]_n$. On déduit qu'il est possible d'étendre l'addition modulaire aux classes d'équivalences. Au lieu d'additionner un élément de $[a]_n$ à un élément de $[b]_n$, nous additionnerons directement ces deux classes d'équivalence. Comme on remarque la même chose en calculant $c = ab \ mod \ n$ (en exercice), nous pouvons définir les calculs modulo $n$ sur l'ensemble d'ensembles

$$\{[0]_n, \ldots, [n-1]_n\}$$

**Définition 15.3.2** *Soit $n > 0$, alors on note $Z/nZ$ l'espace quotient $\{[k]_n | 0 \leq k \leq n-1\}$*

Lorsqu'il n'y a pas d'ambiguïté, on se permet d'enlever les crochets et de noter $k$ au lieu de $[k]_n$.

**Exercice 6**

Montrez qu'en substituant à $a$ et à $b$ des éléments de même classe d'équivalence dans $c = ab \ mod \ n$, on obtient un élément de même classe d'équivalence que $c$.

**Structure d'anneau**

**Propriété 15.3.1** *Pour tout $n > 0$, $Z/nZ$ est un monoïde.*

Preuve en exercice.

**Propriété 15.3.2** *Pour tout $n > 0$, $Z/nZ$ est un groupe.*

Preuve en exercice.

**Propriété 15.3.3** *Pour tout $n > 0$, $Z/nZ$ est un anneau.*

Preuve en exercice.

## 15.4 Inverses modulaires

Supposons qu'il existe une fonction de chiffrement $f$ définie comme suit, $f(k) = k * 7 \ mod\ 9$. Calculons $f(3) \equiv 3 * 7 \equiv 29 \equiv 0 \pmod 9$. Existe-t-il une fonction $f'(k) = k * p \ mod\ 9$, inverse de $f$, telle que $(f \circ f')(k) = k$, en particulier, existe-t-il $p$ tel que $0 * p \ mod\ 9 = 3$ ?

On a multiplié $[3]_9$ par $[7]_9$ pour obtenir $[0]_9$, peut-on "revenir" à $[3]_9$, c'est-à-dire multiplier $[0]_9$ par l'inverse, noté $[7]_9^{-1}$ de $[7]_9$ ? Plus généralement, cela revient à se demander si $Z/nZ$ est un corps. C'est-à-dire, est-ce que la "division" existe dans $Z/nZ$. Voici la réponse :

**Théorème 15.4.1** *$Z/nZ$ est corps si et seulement si $n$ est premier.*

Supposons que $Z/nZ$ est un corps, étant donné $[k]_n$ $(k \neq 0)$, $[k]_n$ admet un inverse. Il existe donc une classe $[k']_n$ telle que $k.k' \equiv 1 \pmod n$. Par définition, $n|(k.k' - 1)$, et il existe $\mu$ tel que

$$\mu n = k.k' - 1$$

ce qui s'ecrit aussi comme une relation de Bezout,

$$k.k' - \mu n = 1$$

Or, $k'$ et $\mu$ n'existent que si $k$ et $n$ sont premiers entre eux. Comme $Z/nZ$ est un corps, alors pour tout $k \in \{1, \ldots, n-1\}$, $k$ est premier avec $n$, donc aucun d'eux (sauf 1) ne peut diviser $n$, donc $n$ est premier. La réciproque est basée sur le même principe.

### 15.4.1 Eléments inversibles et fonction $\phi$ d'Euler

On déduit de la preuve précédente que $k$ est inversible dans $Z/nZ$ si et seulement si $k$ est premier avec $n$. Donc si $n$ n'est pas premier, tous ses éléments ne sont pas inversibles.

**Définition 15.4.1** *Pour tout $n > 0$, $\phi(n)$ est le nombre d'éléments inversibles de $Z/nZ$.*

Prenons par exemple $Z/10Z$, les éléments inversibles de $Z/10Z$ sont

$$\{1, 3, 7, 9\}$$

Notons qu'il s'agit des éléments premiers avec 10. Donc $\phi(10) = 4$.

**Exercice 7**

- $Z/11Z$ est-il un corps ?
- Quels sont les éléments inversibles de $Z/11Z$ et leurs inverses ?
- Quelle est la valeur de $\phi(11)$ ?

**Exercice 8**

- $Z/14Z$ est-il un corps ?
- Quels sont les éléments inversibles de $Z/14Z$ et leurs inverses ?
- Quelle est la valeur de $\phi(14)$ ?

Notons que si $n$ est premier, tous ses éléments, sauf 0 sont inversibles, nous admettrons donc que

**Propriété 15.4.1** *Si $n$ est premier, alors $\phi(n) = n - 1$.*

Nous aurons besoin du résultat suivant pour RSA, admis sans preuve,

**Propriété 15.4.2** *Si $p$ et $q$ sont premiers, alors $\phi(pq) = (p-1)(q-1)$.*

Par exemple, dans $Z/35Z$, il y a $(5-1)(7-1)$ éléments inversibles, donc $\phi(35) = 24$.

### 15.4.2 Pratique de l'inversion

Comment, étant donné $n > 0$, et $k \in Z/nZ$, trouver l'inverse de $k$ ? Si $n$ est premier, le petit théorème de fermat nous le donne :

**Théorème 15.4.2** *Si $n$ est premier, alors pout tout $k$ non congru à 0 modulo $n$,*

$$k^{n-1} \equiv 1 \pmod{n}$$

.

Par exemple, si $n = 5$ et $k = 3$, alors $3^4 \equiv 81 \equiv 1 \pmod 5$, donc $3^{n-2} \equiv 3^3 \equiv 27 \equiv 2 \pmod 5$ est l'inverse modulo 5 de 3. En effet, $3.2 \equiv 1 \pmod 5$.

Si on connait $\phi(n)$ alors le théorème d'Alembert-Gauss nous permet de trouver l'inverse de $k$,

**Théorème 15.4.3** *Pour tous $n \geq 2$, $k$ inversible dans $Z/nZ$,*

$$k^{\phi(n)} \equiv 1 \pmod{n}$$

.

Par exemple, si $n = 10$, et $k = 3$, alors $\phi(10) = \phi(2.5) = 1.4 = 4$, on a bien

$$\begin{aligned} 3^4 &\equiv 9^2 \\ &\equiv 81 \\ &\equiv 1 \pmod{10} \end{aligned}$$

Si $n = 21$ et $k = 4$, alors $\phi(21) = \phi(3.7) = 2.6 = 12$, et on a bien

$$\begin{aligned} 4^{12} &\equiv 16^6 \\ &\equiv 256^3 \\ &\equiv 4^3 \\ &\equiv 64 \\ &\equiv 1 \pmod{10} \end{aligned}$$

L'inverse modulo $pq$ ($p$ et $q$ premiers) de $k$ (inversible dans $Z/pqZ$) est donc $k^{\phi(n)-1}$.

**Exercice 9 - Espaces quotient et GMP**

Ecrire avec GMP un sous-programme faisant le rapport complet d'un espace quotient de la forme $Z/nZ$ :
- $Z/nZ$ est-il un corps ?
- Quels sont les éléments inversibles de $Z/nZ$ et leurs inverses ?
- Quelle est la valeur de $\phi(n)$ ?

# Chapitre 16

# RSA

### 16.0.3 Principe

Le chiffrement se fait à l'aide de la clé publique, le déchiffrement à l'aide de la clé privée. [2]

#### Chiffrement

La clé publique est un couple $(C, N)$ où
- $N = PQ$ est le produit de deux nombres premiers $P$ et $Q$
- $C$ est un nombre premier avec $(P-1)(Q-1)$.

On chiffre un message $M$ en calculant

$$M' \equiv M^C \pmod{N}$$

Si par exemple on a $P = 3$, $Q = 11$, alors $N = PQ = 33$ et $(P-1)(Q-1) = 2 \times 10 = 20$. Si par exemple on prend $C = 7$, on remarque que 7 est bien premier avec 20, la clé publique est $(7, 33)$. On chiffre le message $M = 13$ en calculant $M' \equiv M^C \equiv 13^7 \equiv 7 \pmod{33}$.

#### Déchiffrement

Notez qu'il n'est pas trivial de retrouver $M$ à partir de $M'$ et de $(C, PQ)$. La clé privée est un couple $(C', N)$ tel que pour tout message $M$,

$$(M^C)^{C'} \equiv M \pmod{N}$$

Alors on déchiffre $M'$ en calculant

$$M \equiv (M')^{C'} \pmod{N}$$

Ici on a $C' = 3$. On constate que $M \equiv 7^3 \equiv 13 \pmod{33}$. La question que l'on est invité à se poser est : comment on trouve $C'$ ?

### 16.0.4 Résistance aux attaques

D'après le théorème d'Euler, pour tout $M$ premier avec $N$,

$$M^{\phi(N)} \equiv 1 \pmod{N}$$

Plus généralement, pour tout $k$ entier relatif,

$$M^{k\phi(N)} \equiv 1 \pmod{N}$$

Ce qui s'écrit aussi

$$M^{k\phi(N)} M \equiv M \pmod{N}$$

et de ce fait

$$M^{k\phi(N)+1} \equiv M \pmod{N}$$

Comme

$$M^{CC'} \equiv M \pmod{N}$$

il convient de déterminer $C'$ tel que

$$CC' = k\phi(N) + 1$$

ce qui s'écrit plus simplement comme une relation de Bezout

$$CC' - k\phi(N) = 1$$

Notez que la connaissance de $\phi(N)$ est indispensable si on souhaite trouver $C'$. Etant donné $N = PQ$ le produit de nombres premiers $P$ et $Q$, on a $\phi(N) = (P-1)(Q-1)$. Donc pour connaître $\phi(N)$, il est nécessaire de connaître la décomposition de $N$ en produit de facteurs premiers. Donc, retrouver $C'$ à partir de $C$ oblige le cryptanalyste à passer par une factorisation de $N$, problème connu pour être difficile.

# Annexe A

# Corrigés des programmes

## A.1 Fonctions récursives

```c
#include<stdio.h>
#include<stdlib.h>

int addition(int a, int b)
{
    printf("(%d, %d)\n", a, b);
    if (a == 0)
        return b;
    if (a > 0)
        return 1 + addition(a - 1, b);
    return addition(a + 1, b) - 1;
}

int soustraction(int a, int b)
{
    printf("(%d, %d)\n", a, b);
    if (b == 0)
        return a;
    if (b > 0)
        return soustraction(a, b - 1) - 1;
    return soustraction(a, b + 1) + 1;
}

int multiplication(int a, int b)
{
    printf("(%d, %d)\n", a, b);
    if (a == 0)
        return 0;
    if (a > 0)
        return b + multiplication(a - 1, b);
    return multiplication(a + 1, b) - b;
}

int division(int a, int b)
{
    printf("(%d, %d)\n", a, b);
    if (a < 0)
        return - division(-a, b);
    if (b < 0)
        return - division(a, -b);
    if (a < b)
        return 0;
    return 1 + division(a - b, b);
}

int puissance(int a, int b)
{
    printf("(%d, %d)\n", a, b);
    if (b == 0)
        return 1;
    return a * puissance(a, b - 1);
}
```

## A.2 Fonctions récursives terminales

```c
#include<stdio.h>
#include<stdlib.h>


long factorielleR(long n, long acc)
{
    printf("(%ld, %ld)\n", n, acc);
    if (n == 0)
        return  acc;
    return factorielleR(n-1, acc * n);
}

long factorielle(long n)
{
    return factorielleR(n, 1);
}

int main()
{
    long a = 4;
    printf("%ld ! = %ld\n", a, factorielle(a));
    getch();
    return 0;
}
```

## A.3 Itérateurs et applications

```c
#include<stdio.h>

/*-----------------------------------------*/

typedef struct
{
  unsigned long first;
  unsigned long second;
}couple;

/*-----------------------------------------*/

typedef struct
{
  unsigned long _11, _12, _21, _22;
}quadruplet;

/*-----------------------------------------*/

couple applyF(couple (*f)(couple), int n, couple x)
{
  if (n == 0)
    return x;
  return f(applyF(f, n - 1, x));
}

/*-----------------------------------------*/

couple applyFT(couple (*f)(couple), int n, couple x)
{
  if (n == 0)
    return x;
  return applyF(f, n-1, f(x));
}

/*-----------------------------------------*/

unsigned long multiplie(unsigned long a, unsigned long b)
{
  couple x = {0, a};
  couple f(couple x)
    {
      couple y = {x.first + x.second, x.second};
      return y;
    }
  return applyFT(f, b, x).first;
}

/*-----------------------------------------*/

unsigned long factorielle(unsigned long n)
{
  couple x = {n, 1};
  couple f(couple x)
    {
      couple y = {x.first - 1, x.first*x.second};
      return y;
    }
  return applyFT(f, n, x).second;
}

/*-----------------------------------------*/

unsigned long puissance(unsigned long b, unsigned long n)
{
  couple x = {1, b};
  couple f(couple x)
    {
      couple y = {x.first * x.second, x.second};
      return y;
    }
  return applyFT(f, n, x).first;
}

/*-----------------------------------------*/

unsigned long fastPuissance(unsigned long b, unsigned long n)
{
  if(n == 0)
```

```
    return 1;
  if (n % 2)
      return b * fastPuissance(b*b, n/2);
  return fastPuissance(b*b, n/2);
}

/*----------------------------------------*/

unsigned long applyFD(int (*delta)(int k),
                      unsigned long (*f)(couple), int n, unsigned long x)
{
  couple y;
  if (n == 0)
    return x;
  y.first = n;
  y.second = applyFD(delta, f, delta(n), x);
  return f(y);
}

/*----------------------------------------*/

unsigned long slowPuissanceIT(unsigned long b, unsigned long n)
{
  unsigned long f(couple x)
    {
      return  x.second*b;
    }
  int delta(int x)
    {
      return x - 1;
    }
  return applyFD(delta, f, n, 1);
}

/*----------------------------------------*/

unsigned long factorielleIT(unsigned long n)
{
  unsigned long f(couple x)
    {
      return  x.first * x.second;
    }
  int delta(int x)
    {
      return x - 1;
    }
  return applyFD(delta, f, n, 1);
}

/*----------------------------------------*/

unsigned long fastPuissanceIT(unsigned long b, unsigned long n)
{
  unsigned long f(couple x)
    {
      if (x.first % 2)
        return b*x.second*x.second;
      return x.second*x.second;
    }
  int delta(int x)
    {
      return x/2;
    }
  return applyFD(delta, f, n, 1);
}

/*----------------------------------------*/

quadruplet fiboMat()
{
  quadruplet q = {1, 1, 1, 0};
  return q;
}

/*----------------------------------------*/

quadruplet identite()
{
  quadruplet q = {1, 0, 0, 1};
  return q;
}
```

```
/*----------------------------------------*/

quadruplet multMat(quadruplet x, quadruplet y)
{
  quadruplet xy;
  xy._11 = x._11*y._11 + x._12*y._21;
  xy._21 = x._21*y._11 + x._22*y._21;
  xy._12 = x._11*y._12 + x._12*y._22;
  xy._22 = x._21*y._12 + x._22*y._22;
  return xy;
}

/*----------------------------------------*/

quadruplet applyFDQ(unsigned long (*delta)(unsigned long),
                    quadruplet (*f)(unsigned long, quadruplet),
                    unsigned long n, quadruplet x)
{
  if (n == 0)
    return x;
  return f(n, applyFDQ(delta, f, delta(n), x));
}

/*----------------------------------------*/

quadruplet matPow(quadruplet x, unsigned long n)
{
  quadruplet f(unsigned long m, quadruplet q)
    {
      quadruplet qPrime = multMat(q, q);
      if (m % 2)
        return multMat(x, qPrime);
      return qPrime;
    }
  unsigned long delta(unsigned long x)
    {
      return x/2;
    }
  return applyFDQ(delta, f, n, identite());
}

/*----------------------------------------*/

couple multMatVec(quadruplet x, couple y)
{
  couple xy;
  xy.first = x._11*y.first + x._12*y.second;
  xy.second = x._21*y.first + x._22*y.second;
  return xy;
}

/*----------------------------------------*/

unsigned long fastFibonacciIT(unsigned long l)
{
  quadruplet q = fiboMat();
  couple c = {1, 0};
  c = multMatVec(matPow(q, l), c);
  return c.second;
}

/*----------------------------------------*/

int main()
{
  int i;
  for(i = 0 ; i <= 45 ; i++)
    {
      printf("F_%d = %ld\n", i, fastFibonacciIT(i));
    }
 return 0;
}
```

## A.4 Tours de Hanoï

```c
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

/*
  On placera dans la premiere case de chaque tableau le nombre de disques
  places sur le socle.
*/

int max2(int i, int j)
{
  return  (i > j) ? i : j;
}

int max3(int i, int j, int k)
{
  return max2(max2(i, j), k);
}

void afficheSocles(int** s)
{
  int max = max3(**s, **(s + 1), **(s + 2));
  int i;
  for (i = max ; i > 0 ; i--)
    {
      if (**s >= i)
        printf("%3d ", *(*s + i));
      else
        printf("    ");
      if (**(s + 1) >= i)
        printf("%3d ", *(*(s+1) + i));
      else
        printf("    ");
      if (**(s + 2) >= i)
        printf("%3d ", *(*(s+2) + i));
      printf("\n");
    }
  printf("-------------\n");
}

void deplaceDisque(int* socleSource, int* socleDestination)
{
  *(socleDestination + *socleDestination + 1) =
    *(socleSource + *socleSource);
  (*socleDestination)++;
  (*socleSource)--;
}

void deplaceDisques(int n, int** s, int sFrom, int sVia, int sTo)
{
  if (n > 0)
    {
      deplaceDisques(n-1, s, sFrom, sTo, sVia);
      deplaceDisque(*(s + sFrom), *(s + sTo));
      afficheSocles(s);
      deplaceDisques(n-1, s, sVia, sFrom, sTo);
    }
}


int* creeSocle(int nbDisques)
{
  int* s = malloc((nbDisques + 1)*sizeof(int));
  if (s == NULL)
    exit(0);
  *s = 0;
  return s;
}

void initSocle(int* s, int nbDisques)
{
  int i;
  for(i = 1 ; i <= nbDisques ; i++)
      *(s + i) = nbDisques - i + 1;
  *s = nbDisques;
}

void creeSocles(int** s, int nbDisques)
{
```

```
  *s = creeSocle(nbDisques);
  *(s + 1) = creeSocle(nbDisques);
  *(s + 2) = creeSocle(nbDisques);
  initSocle(*s, nbDisques);
}

int main()
{
  int* s[3];
  int nbDisques = 3;
  creeSocles(s, nbDisques);
  afficheSocles(s);
  deplaceDisques(nbDisques, s, 0, 1, 2);
  free(*s);
  free(*(s + 1));
  free(*(s + 2));
  return 1;
}
```

## A.5 Suite de Fibonacci

```c
#include<stdio.h>

#define N 45

unsigned long fibo(unsigned long l)
{
  if (l == 0)
    return 0;
  if (l == 1)
    return 1;
  return fibo(l - 1) + fibo(l - 2);
}

void phi(unsigned long* l1, unsigned long* l2)
{
  unsigned long temp;
  temp = *l1;
  *l1 += *l2;
  *l2 = temp;
}

void applyF(void (*f)(unsigned long*, unsigned long*),
            int n, unsigned long* l1, unsigned long* l2)
{
  if (n > 0)
    {
      f(l1, l2);
      applyF(f, n-1, l1, l2);
    }
}

unsigned long fastFibo(unsigned long l)
{
  unsigned long l1 = 1, l2 = 0;
  applyF(phi, l, &l1, &l2);
  return l2;
}

int main()
{
  unsigned long l;
  for(l = 0 ; l <= N ; l++)
    printf("%lu = %lu\n", l, fibo(l));
  for(l = 0 ; l <= N ; l++)
    printf("%lu = %lu\n", l, fastFibo(l));
  return 0;
}
```

## A.6 Pgcd

```
#include<stdio.h>

unsigned long pgcd(unsigned long a, unsigned long b)
{
  if (b == 0)
    return a;
  return pgcd(b, a % b);
}

unsigned long fibonacci(unsigned long l)
{
  unsigned long moinsUn, moinsDeux;
  if (l == 0 || l == 1)
    return l;
  moinsUn = 1;
  moinsDeux = 0;
  while( l > 1)
    {
      moinsUn ^= moinsDeux;
      moinsDeux ^= moinsUn;
      moinsUn ^= moinsDeux;
      moinsUn += moinsDeux;
      l--;
    }
  return moinsUn;
}

int main()
{
  unsigned long f1, f2, i;
  f2 = fibonacci(0);
  for(i = 0 ; i <= 40 ; i++)
    {
      f1 = f2;
      f2 = fibonacci(i+1);
      printf("pgcd(%lu, %lu) = %lu\n", f1, f2, pgcd(f1, f2));
    }
  return 0;
}
```

## A.7 Pavage avec des L

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>

typedef struct
{
  int ligne;
  int colonne;
}coordonnees;

typedef struct
{
  char* tab;
  int n;
  coordonnees caseInterdite;
}grille;

int puiss(int b, int n)
{
  if (n == 0)
    return 1;
  return b * puiss(b, n-1);
}

void erreurMemoire()
{
  printf("Pas de memoire\n");
  exit(1);
}


char* retourneAdresseCase(grille g, coordonnees c)
{
  int milieu;
  if (g.n == 0)
    return g.tab;
  milieu = puiss(2, g.n - 1);
  if (c.ligne >= milieu)
    {
      g.tab += 2*milieu*milieu;
      c.ligne-=milieu;
    }
  if (c.colonne >= milieu)
    {
      g.tab += milieu*milieu;
      c.colonne-=milieu;
    }
  g.n -= 1;
  return retourneAdresseCase(g, c);
}

char retourneValeurCase(grille g, coordonnees c)
{
  return *retourneAdresseCase(g, c);
}

void affecteValeurCase(grille g, coordonnees c, char valeur)
{
  *retourneAdresseCase(g, c) = valeur;
}

grille grilleCree(int n, int ligneInterdite, int colonneInterdite,
                  char valeurs, char valeurInterdite)
{
  grille g;
  int largeur = puiss(2, n);
  coordonnees c;
  g.tab = (char*)malloc(largeur * largeur);
  if (g.tab == NULL)
    erreurMemoire();
  g.n = n;
  g.caseInterdite.ligne = ligneInterdite;
  g.caseInterdite.colonne = colonneInterdite;
  for(c.ligne = 0 ; c.ligne < largeur ; c.ligne++)
      for(c.colonne = 0 ; c.colonne < largeur; c.colonne++)
            affecteValeurCase(g, c, valeurs);
  affecteValeurCase(g, g.caseInterdite, valeurInterdite);
  return g;
}
```

```
void grilleAffiche(grille g)
{
  int largeur = puiss(2, g.n);
  coordonnees c;
  for(c.ligne = 0 ; c.ligne < largeur ; c.ligne++)
    {
      for(c.colonne = 0 ; c.colonne < largeur; c.colonne++)
          printf("%c ", retourneValeurCase(g, c));
      printf("\n");
    }
}

void grillePavage(grille g, char c)
{
  int largeur;
  char valeurInterdite;
  int nbPieces;
  grille hd, hg, bd, bg;
  if (g.n != 0)
    {
      largeur = puiss(2, g.n-1);
      hg.tab = g.tab;
      hd.tab = hg.tab + largeur*largeur;
      bg.tab = hd.tab + largeur*largeur;
      bd.tab = bg.tab + largeur*largeur;
      hg.n = g.n - 1;
      hd.n = g.n - 1;
      bg.n = g.n - 1;
      bd.n = g.n - 1;
      hg.caseInterdite.ligne = largeur - 1;
      hg.caseInterdite.colonne = largeur - 1;
      hd.caseInterdite.ligne = largeur - 1;
      hd.caseInterdite.colonne = 0 ;
      bg.caseInterdite.ligne = 0 ;
      bg.caseInterdite.colonne = largeur - 1;
      bd.caseInterdite.ligne = 0 ;
      bd.caseInterdite.colonne = 0 ;
      if (g.caseInterdite.ligne < largeur)
          if (g.caseInterdite.colonne < largeur )
              hg.caseInterdite = g.caseInterdite;
          else
            {
              hd.caseInterdite.ligne = g.caseInterdite.ligne ;
              hd.caseInterdite.colonne = g.caseInterdite.colonne - largeur;
            }
      else
          if (g.caseInterdite.colonne < largeur )
            {
              bg.caseInterdite.ligne = g.caseInterdite.ligne - largeur ;
              bg.caseInterdite.colonne = g.caseInterdite.colonne - largeur;
            }
          else
            {
              bd.caseInterdite.ligne = g.caseInterdite.ligne - largeur ;
              bd.caseInterdite.colonne = g.caseInterdite.colonne - largeur;
            }
      valeurInterdite = retourneValeurCase(g, g.caseInterdite);
      affecteValeurCase(hd, hd.caseInterdite, c + 1);
      affecteValeurCase(hg, hg.caseInterdite, c + 1);
      affecteValeurCase(bg, bg.caseInterdite, c + 1);
      affecteValeurCase(bd, bd.caseInterdite, c + 1);
      affecteValeurCase(g, g.caseInterdite, valeurInterdite);
      grillePavage(hd, c+1);
      nbPieces = (largeur*largeur - 1)/3;
      grillePavage(hg, c+nbPieces + 1);
      grillePavage(bg, c+2*nbPieces + 1);
      grillePavage(bd, c+3*nbPieces + 1);
    }
}

void grilleDetruit(grille g)
{
  free(g.tab);
}

int main()
{
  grille g = grilleCree(4, 5, 12, -1, '!');
  grillePavage(g, '!');
  grilleAffiche(g);
```

```
    grilleDetruit(g);
    return 0;
}
```

## A.8 Complexes

```c
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>
#include"complexe.h"
#include"linkedList.h"

/*-------------------------------------------*/

/*
  Retourne la partie reelle
 */

double reComplexe(complexe c)
{
  return c.re;
}

/*-------------------------------------------*/

/*
  Retourne la partie imaginaire
 */

double imComplexe(complexe c)
{
  return c.im;
}

/*-------------------------------------------*/

/*
  Retourne le module
 */

double modComplexe(complexe c)
{
  return c.mod;
}

/*-------------------------------------------*/

/*
  Retourne l'argument
 */

double argComplexe(complexe c)
{
  return c.arg;
}

/*-------------------------------------------*/

/*
  Affiche c sous forme exponentielle
 */

void printExpComplexe(complexe c)
{
  printf("%.10lfe^(%.10lfi)", c.mod, c.arg);
}

/*-------------------------------------------*/

/*
  Affiche l'affixe de c
 */

void printAffComplexe(complexe c)
{
  printf("(%.10lf + %.10lfi)", c.re, c.im);
}

/*-------------------------------------------*/

/*
  Cree un complexe a partir de son affixe
 */
```

```
complexe makeAffComplexe(double re, double im)
{
  complexe c = {re, im, sqrt(re*re + im*im), 0.0};
  if (c.mod != 0.0)
    {
      c.arg = acos(c.re/c.mod) * ((c.im >= 0 )? 1 : -1) ;
    }
  return c;
}

/*-----------------------------------------*/

/*
  Cree un complexe a partir de sa forme exponentielle
 */

complexe makeExpComplexe(double mod, double arg)
{
  complexe c = {0.0, 0.0, mod, arg};
  c.arg += M_PI;
  long l = c.arg/(2.*M_PI);
  c.arg  -= l*2.*M_PI;
  c.arg -= M_PI;
  if (mod != 0.0)
    {
      c.re = c.mod * cos(c.arg);
      c.im = c.mod * sin(c.arg);
    }
  return c;
}

/*-----------------------------------------*/

pComplexe copyComplexe(complexe c)
{
  pComplexe p = (pComplexe)malloc(sizeof(complexe));
  if (p == NULL)
    exit(0);
  p->re = c.re;
  p->im = c.im;
  p->mod = c.mod;
  p->arg = c.arg;
  return p;
}

/*-----------------------------------------*/

/*
  Retourne le conjugue de c
 */

complexe conjugueComplexe(complexe c)
{
  return makeAffComplexe(c.re, -c.im);
}

/*-----------------------------------------*/

/*
  Soustrait c1 et c2
 */

complexe subComplexe(complexe c1, complexe c2)
{
  return makeAffComplexe(c1.re - c2.re, c1.im - c2.im);
}

/*-----------------------------------------*/

/*
  Additionne c1 et c2
 */

complexe addComplexe(complexe c1, complexe c2)
{
  return makeAffComplexe(c1.re + c2.re, c1.im + c2.im);
}

/*-----------------------------------------*/

/*
```

```
  Multiplie c1 et c2
 */

complexe multComplexe(complexe c1, complexe c2)
{
  return makeAffComplexe(c1.re * c2.re - c1.im * c2.im,
                         c1.re * c2.im + c1.im * c2.re);
}

/*----------------------------------------*/

/*
  Multiplie  c par le reel d
 */

complexe multReelComplexe(double d, complexe c)
{
  return makeExpComplexe(d * c.mod, c.arg);
}

/*----------------------------------------*/

/*
  Divise c2 par c1
 */

complexe divComplexe(complexe c1, complexe c2)
{
  return makeExpComplexe(c1.mod / c2.mod, c1.arg - c2.arg);
}

/*----------------------------------------*/

/*
  Eleve c a la puissane n
 */

complexe puissComplexe(complexe c, unsigned long n)
{
  return makeExpComplexe(pow(c.mod, n), n*c.arg);
}
```

## A.9 Listes chaînées

```
#include"linkedList.h"
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>

/*----------------------------------------*/

void error()
{
  printf("error");
  exit(0);
}

/*----------------------------------------*/

int linkedListGetSize(linkedList* l)
{
  return l->size;
}

/*----------------------------------------*/


linkedList* linkedListCreate()
{
  linkedList* l;
  l = (linkedList*)malloc(sizeof(linkedList));
  if(l == NULL)
    error();
  l->first = NULL;
  l->last = NULL;
  l->size = 0;
  return l;
}

/*----------------------------------------*/


link* linkedListCreateLink(void* x)
{
  link* lk = (link*)malloc(sizeof(link));
  if (lk == NULL) error();
  lk->data = x;
  lk->next = NULL;
  return lk;
}

/*----------------------------------------*/


void linkedListAppendLink(linkedList* l, link* lk)
{
  if (l->last == NULL)
    {
      l->first = lk;
      l->last = l->first;
    }
  else
    {
      l->last->next = lk;
      l->last = lk;
    }
  lk->next = NULL;
  l->size ++;
}

/*----------------------------------------*/


void linkedListAppend(linkedList* l, void* x)
{
  link* lk = linkedListCreateLink(x);
  linkedListAppendLink(l, lk);
}

/*----------------------------------------*/


void linkedListPushLink(linkedList* l, link* lk)
```

```
{
  if (l->first == NULL)
    {
      l->first = lk;
      l->last = l->first;
    }
  else
    {
      lk->next = l->first;
      l->first = lk;
    }
  l->size ++;
}

/*----------------------------------------*/


void linkedListPush(linkedList* l, void* x)
{
  link* lk = linkedListCreateLink(x);
  linkedListPushLink(l, lk);
}

/*----------------------------------------*/


link* linkedListGetFirst(linkedList* l)
{
  return l->first;
}

/*----------------------------------------*/


link* linkedListUnlinkFirst(linkedList* l)
{
  link* f = linkedListGetFirst(l);
  if (f != NULL)
    {
      l->first = l->first->next;
      if (l->last == f)
        l->last = NULL;
      l->size--;
    }
  return f;
}

/*----------------------------------------*/

void linkedListConcat(linkedList* begin, linkedList* end)
{
  if (begin->size != 0)
      begin->last->next = end->first;
  else
      begin->first = end->first;
  if(end->size != 0)
    begin->last = end->last;
  end->first = NULL;
  end->last = NULL;
  begin->size += end->size;
  end->size = 0;
}

/*----------------------------------------*/

linkedList* linkedListMap(linkedList* l, void* (*f)(void*))
{
  linkedList* lBis = linkedListCreate();
  link* lk = linkedListGetFirst(l);
  while(lk!=NULL)
    {
      linkedListAppend(lBis, f(lk->data));
      lk = lk->next;
    }
  return lBis;
}

/*----------------------------------------*/

void linkedListApply(linkedList* l, void (*f)(void*))
{
```

```
  link* lk = linkedListGetFirst(l);
  while(lk!=NULL)
    {
      f(lk->data);
      lk = lk->next;
    }
}

/*----------------------------------------*/

void linkDestroy(link* l, void (*fr)(void*))
{
  if (l != NULL)
    {
      linkDestroy(l->next, fr);
      fr(l->data);
      free(l);
    }
}

/*----------------------------------------*/

void linkedListDestroy(linkedList* l, void (*fr)(void*))
{
  linkDestroy(l->first, fr);
  free(l);
}

/*----------------------------------------*/



/*
void destroyInt(void* i)
{
  free((int*)i);
}

void printInt(void* i)
{
  printf("%d -> ", *((int*)i));
}

void* copyInt(void* i)
{
  void* v = malloc(sizeof(int));
  if (v == NULL)
    error();
  *((int*)v) = *((int*)i);
  return v;
}

int main()
{
  int i = 1;
  int* k;
  linkedList* l = linkedListCreate();
  linkedList* lBis;
  for(i = 0 ; i < 100 ; i++)
    {
      k = (int*)malloc(sizeof(int));
      *k = i;
      linkedListAppend(l, k);
    }
  //linkedListApply(l, printInt);
  lBis = linkedListMap(l, copyInt);
  //linkedListApply(lBis, printInt);
  linkedListConcat(l, lBis);
  linkedListApply(l, printInt);
  linkedListDestroy(l, destroyInt);
  linkedListDestroy(lBis, destroyInt);
  return 1;
}

*/
```

## A.10 Tri fusion

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include"linkedList.h"

void split(linkedList* source, linkedList* dest1, linkedList* dest2)
{
  link* l = linkedListUnlinkFirst(source);
  if(l != NULL)
    {
      linkedListAppendLink(dest1, l);
      split(source, dest2, dest1);
    }
}

void merge(linkedList* source1, linkedList* source2, linkedList* dest,
           int (*inf)(void*, void*))
{
  link *l1, *l2;
  l1 = linkedListGetFirst(source1);
  l2 = linkedListGetFirst(source2);
  if (l1 != NULL || l2 != NULL)
    {
      if (l2 == NULL ||
          (l1 != NULL && inf(l1->data, l2->data))
          )
        {
          linkedListUnlinkFirst(source1);
          linkedListAppendLink(dest, l1);
        }
      else
        {
          linkedListUnlinkFirst(source2);
          linkedListAppendLink(dest, l2);
        }
      merge(source1, source2, dest, inf);
    }
}

void doNothing(void* v)
{
}

void triFusion(linkedList* l, int (*inf)(void*, void*))
{
  linkedList *l1, *l2;
  if (linkedListGetSize(l) > 1)
    {
      l1 = linkedListCreate();
      l2 = linkedListCreate();
      split(l, l1, l2);
      triFusion(l1, inf);
      triFusion(l2, inf);
      merge(l1, l2, l, inf);
      linkedListDestroy(l1, doNothing);
      linkedListDestroy(l2, doNothing);
    }
}

/*-------------------------------------------------------------*/

int estInf(void* i, void* j)
{
  return (*(int*)i <= *(int*)j  ) ? 1 : 0;
}

void destroyInt(void* i)
{
  free((int*)i);
}

int main()
{
  int i = 1;
  int* k;
  linkedList* l = linkedListCreate();
  link* lk;
  for(i = 100 ; i > 0 ; i--)
    {
```

```
      k = (int*)malloc(sizeof(int));
      *k = i;
      linkedListAppend(l, k);
    }
  lk = linkedListGetFirst(l);
  while(lk!=NULL)
    {
      printf("%d ", *((int*)(lk->data)));
      lk = lk->next;
    }
  printf("\n");
  triFusion(l, estInf);
  lk = linkedListGetFirst(l);
  while(lk!=NULL)
    {
      printf("%d ", *((int*)(lk->data)));
      lk = lk->next;
    }
  printf("\n");
  linkedListDestroy(l, destroyInt);
  return 0;
}
```

## A.11 Tri par insertion

```c
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include"linkedList.h"

link* insert(link* list, link* item, int (*estInf)(void*, void*))
{
  if (list == NULL)
    {
      item->next = NULL;
      return item;
    }
  if (estInf(item->data, list->data))
    {
      item->next = list;
      return item;
    }
  list->next = insert(list->next, item, estInf);
  return list;
}

link* triInsertion(link* list, int (*estInf)(void*, void*))
{
  if (list == NULL)
    return NULL;
  return insert(triInsertion(list->next, estInf), list, estInf);
}

void linkedListSort(linkedList* l, int (*estInf)(void*, void*))
{
  l->first = triInsertion(l->first, estInf);
  void setLast(link* ll)
    {
      if (ll == NULL)
        l->last = NULL;
      else
        if (ll->next == NULL)
          l->last = ll;
        else
          setLast(ll->next);
    }
  setLast(l->first);
}

int estInf(void* i, void* j)
{
  return (*(int*)i <= *(int*)j  ) ? 1 : 0;
}

void destroyInt(void* i)
{
  free((int*)i);
}

int main()
{
  int i = 1;
  int* k;
  linkedList* l = linkedListCreate();
  for(i = 100 ; i > 0 ; i--)
    {
      k = (int*)malloc(sizeof(int));
      *k = i;
      linkedListAppend(l, k);
    }
  void printInt(void* i)
    {
      printf("%d ", *((int*)i));
    }
  linkedListApply(l, printInt);
  printf("\n");
  linkedListSort(l, estInf);
  linkedListApply(l, printInt);
  printf("\n");
  linkedListDestroy(l, destroyInt);
  return 0;
}
```

## A.12 Transformée de Fourier

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>
#include"complexe.h"
#include"fourier.h"
#include"linkedList.h"

/*----------------------------------------*/

/*
  Retourne les racines n-eme de c, O(n)
 */

linkedList* racinesComplexe(complexe c, unsigned long n)
{
  linkedList* l = linkedListCreate();
  pComplexe p;
  unsigned long i;
  for(i = 0 ; i < n ; i++)
    {
      p = (pComplexe)malloc(sizeof(complexe));
      if (p == NULL)
        exit(1);
      *p = makeExpComplexe(pow(c.mod, 1./n), (c.arg/n) + (i)*2*M_PI/n);
      linkedListAppend(l, p);
    }
  return l;
}

/*----------------------------------------*/

/*
  Retourne les racines n-eme de 1, O(n)
 */

linkedList* uniteRacinesComplexe(unsigned long n)
{
  return racinesComplexe(makeAffComplexe(1.0, 0.0), n);
}

/*----------------------------------------*/

/*
  Donne l'image de x par polynome. Les coefficients du polynome sont
  ranges par ordre de degre croissant. O(n).
 */

complexe horner(link* polynome, complexe x)
{
  if (polynome == NULL)
    return makeAffComplexe(0.0, 0.0);
  return addComplexe(*(pComplexe)polynome->data,
                     multComplexe(x,
                                  horner(polynome->next,
                                         x)));
}

/*----------------------------------------*/

/*
  Detruit complexe pointe par p.
*/

void deallocateComplexe(void* p)
{
  free((pComplexe)p);
}

/*----------------------------------------*/

/*
  Retourne la transformee discrete de Fourier du vecteur l,
  version iterative en O(n^2)
 */

linkedList* transformeeFourierIt(linkedList* polynome)
{
  linkedList* FT = linkedListCreate();
```

```
  linkedList* W_n = uniteRacinesComplexe(linkedListGetSize(polynome));
  void* evaluate(void* w)
    {
      pComplexe p = (pComplexe)malloc(sizeof(complexe));
      if (p == NULL)
        exit(0);
      *p = horner(linkedListGetFirst(polynome),
                  *(pComplexe)w);
      return p;
    }
  FT = linkedListMap(W_n, evaluate);
  linkedListDestroy(W_n, deallocateComplexe);
  return FT;
}

/*----------------------------------------*/

/*
  Ne fait rien...
*/

void doNothing(void* w)
{
}

/*----------------------------------------*/

/*
  Retourne la transformee discrete de Fourier du vecteur l,
  version recursive en O(n.log n)
 */

linkedList* FFT(linkedList* polynome)
{
  // creation des listes
  linkedList* FT = linkedListCreate(); // resultat
  linkedList* odd =  linkedListCreate(); // coefficients d'indice pair
  linkedList* even =  linkedListCreate();// coefficients d'indice impair
  linkedList* a_0 =  linkedListCreate(); // TF de odd
  linkedList* a_1 =  linkedListCreate(); // TF de even
  link* l; // pour parcourir FT
  complexe w = makeAffComplexe(1.0, 0.0);
  complexe c = makeExpComplexe(1.0, 2*M_PI/linkedListGetSize(polynome));
  int isEven = 1;
  // cas de base
  if (linkedListGetSize(polynome) == 1)
    {
      linkedListAppend(FT,
                       copyComplexe(*(pComplexe)
                                    (linkedListGetFirst(polynome)->data)));
      return FT;
    }
  // autres cas
  void split(void* coeff)
    {
      if (isEven)
        linkedListAppend(even, coeff);
      else
        linkedListAppend(odd, coeff);
      isEven = !isEven;
    }
  // separation de polynome
  linkedListApply(polynome, split);
  // appels recursifs
  a_0 = FFT(even);
  a_1 = FFT(odd);
  void evenValues(void* v)
    {
      linkedListAppend(FT, copyComplexe(*(pComplexe)v));
    }
  // ajout dans FT des valeurs de a^[0]
  linkedListApply(a_0, evenValues);
  linkedListApply(a_0, evenValues);
  l = linkedListGetFirst(FT);
  void oddValues(void* v)
    {
      pComplexe ft = (pComplexe)(l->data);
      *ft = addComplexe(*ft, multComplexe(w, *(pComplexe)v));
      w = multComplexe(c, w);
      l = l->next;
    }
```

```
  // ajout dans FT des w_n^p * a^[1]_p
  linkedListApply(a_1, oddValues);
  linkedListApply(a_1, oddValues);
  // liberation de la memoire
  linkedListDestroy(odd, doNothing);
  linkedListDestroy(even, doNothing);
  linkedListDestroy(a_0, deallocateComplexe);
  linkedListDestroy(a_1, deallocateComplexe);
  return FT;
}

/*----------------------------------------*/

/*
  Retourne la transformee discrete de Fourier du vecteur l,
  version recursive en O(n.log n)
 */

linkedList* inverseFFT(linkedList* polynome)
{
  linkedList* FT = linkedListCreate();
  linkedList* odd =  linkedListCreate();
  linkedList* even =  linkedListCreate();
  linkedList* a_0 =  linkedListCreate();
  linkedList* a_1 =  linkedListCreate();
  link* l;
  complexe w = makeAffComplexe(1.0, 0.0);
  complexe c = makeExpComplexe(1.0, -2.*M_PI/linkedListGetSize(polynome));
  int isEven = 1;
  if (linkedListGetSize(polynome) == 1)
    {
      linkedListAppend(FT,
                       copyComplexe(*(pComplexe)
                                    (linkedListGetFirst(polynome)->data)));
      return FT;
    }
  void split(void* coeff)
    {
      if (isEven)
        linkedListAppend(even, coeff);
      else
        linkedListAppend(odd, coeff);
      isEven = !isEven;
    }
  linkedListApply(polynome, split);
  a_0 = inverseFFT(even);
  a_1 = inverseFFT(odd);
  void evenValues(void* v)
    {
      linkedListAppend(FT, copyComplexe(*(pComplexe)v));
    }
  linkedListApply(a_0, evenValues);
  linkedListApply(a_0, evenValues);
  l = linkedListGetFirst(FT);
  void oddValues(void* v)
    {
      pComplexe ft = (pComplexe)(l->data);
      complexe value  = multComplexe(w, *(pComplexe)v);
      value = multReelComplexe(1./2., addComplexe(*ft, value));
      *ft = value;
      w = multComplexe(c, w);
      l = l->next;
    }
  linkedListApply(a_1, oddValues);
  linkedListApply(a_1, oddValues);
  void doNothing(void* w){}
  linkedListDestroy(odd, doNothing);
  linkedListDestroy(even, doNothing);
  void deallocateComplexe(void* p)
    {
      free((pComplexe)p);
    }
  linkedListDestroy(a_0, deallocateComplexe);
  linkedListDestroy(a_1, deallocateComplexe);
  return FT;
}

/*----------------------------------------*/

/*
  Retourne la transformee discrete de Fourier inverse
```

```
  du vecteur l, version iterative en O(n^2)
 */

linkedList* transformeeInverseFourierIt(linkedList* polynome)
{
  linkedList* FT = linkedListCreate();
  int n = linkedListGetSize(polynome);
  linkedList* W_n = uniteRacinesComplexe(n);
  void conjugue(void* w)
    {
      *(pComplexe)w = conjugueComplexe(*(pComplexe)w);
    }
  linkedListApply(W_n, conjugue);
  void* evaluate(void* w)
    {
      pComplexe p = (pComplexe)malloc(sizeof(complexe));
      if (p == NULL)
        exit(0);
      *p = horner(linkedListGetFirst(polynome),
                  *(pComplexe)w);
      return p;
    }
  FT = linkedListMap(W_n, evaluate);
  void divise(void* w)
    {
      *(pComplexe)w = multReelComplexe(1./n, *(pComplexe)w);
    }
  linkedListApply(FT, divise);
  void deallocateComplexe(void* p)
    {
      free((pComplexe)p);
    }
  linkedListDestroy(W_n, deallocateComplexe);
  return FT;
}

/*-----------------------------------------*/

linkedList* simpleList(int n)
{
  if (n == 0)
    return linkedListCreate();
  linkedList* l =  simpleList(n - 1);
  linkedListAppend(l, copyComplexe(makeAffComplexe(n, 0.0)));
  return l;
}

/*-----------------------------------------*/
/*
int main()
{
  int k = 512;
  void printC(void* v)
    {
      printAffComplexe(*((pComplexe)v));
      printf("\n");
    }
  printf("liste\n");
  linkedListApply(simpleList(k),
                  printC);
  printf("FFT de liste\n");
  linkedListApply(FFT(simpleList(k)),
                  printC);
  printf("FFT-1 o FFT de liste\n");
  linkedListApply(inverseFFT(FFT(simpleList(k))),
                  printC);
  return 0;
}
*/
```

## A.13 Polynômes

```c
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>
#include"polynomes.h"

/*-------------------------------------------------------*/

/*
Retourne un polynome vide.
 */

linkedList* makePolynome()
{
  return linkedListCreate();
}

/*-------------------------------------------------------*/

/*
  Alloue dynamiquement un emplacement pour une variable de type
  double, y affecte d et retourne cette adresse.
 */

double* makeDouble(double d)
{
  double* p = (double*)malloc(sizeof(double));
  if (p == NULL)
    exit(0);
  *p = (double)d;
  return p;
}

/*-------------------------------------------------------*/

/*
  Alloue dynamiquement un emplacement pour une variable de type
  double, y affecte d et retourne cette adresse.
 */

void* makeDoubleWithVoid(void* v)
{
  return makeDouble(*(double*)v);
}

/*-------------------------------------------------------*/

/*
  Desalloue une variable de type double.
 */

void destroyDouble(void* v)
{
  free((double*)v);
}

/*-------------------------------------------------------*/

/*
  Retourne une copie de l, copie aussi les coefficients.
*/

linkedList* copyPolynome(linkedList* l)
{
  return linkedListMap(l, makeDoubleWithVoid);
}

/*-------------------------------------------------------*/

/*
  Detruit le polynome l et ses coefficients.
 */

void destroyPolynome(linkedList* l)
{
  linkedListDestroy(l, destroyDouble);
}

/*-------------------------------------------------------*/
```

```
/*
  Affiche le monome coeff * x^degre proprement, firstPrinted
  est vrai si et seulement si ce monome est le premier affiche
  dans le polynome.
 */

int printMonome(double coeff, int degre, int firstPrinted)
{
  if (coeff != 0.0)
    {
      if (firstPrinted && coeff > 0.0)
          printf(" + ");
      if(coeff<0.0)
        printf(" - ");
      printf("%4.2lf", (coeff>0)?coeff:-coeff);
      if(degre > 0)
        printf("x");
      if(degre > 1)
        printf("^");
      if(degre > 1)
        printf("%d", degre);
      return 1;
    }
  return 0;
}

/*-------------------------------------------------------*/

/*
  Affiche polynome le plus proprement possible.
*/

void printPolynome(linkedList* polynome)
{
  int degre = 0, firstPrinted = 0;
  void f(void* coeff)
    {
      if (printMonome(*(double*)coeff, degre, firstPrinted))
        firstPrinted = 1;
      degre++;
    }
  linkedListApply(polynome, f);
  printf("\n");
}

/*-------------------------------------------------------*/

/*
  Supprime les O non significatifs Ã  la fin du polynÃ´me.
 */

void skipZerosPolynome(linkedList *l)
{
}

/*-------------------------------------------------------*/

/*
  Retourne le degre de l.
 */

int degreePolynome(linkedList *l)
{
  int n;
  skipZerosPolynome(l);
  n = linkedListGetSize(l);
  if(n != 1)
    return n;
  return (*(double*)linkedListGetFirst(l)->data != 0) - 1;
}

/*-------------------------------------------------------*/

/*
  Ajoute un monome de coefficient coeff a la fin du polynome, faisant
  ainsi croutre son degre de 1.
*/

void appendCoeffPolynome(linkedList* polynome, double coeff)
{
```

```
  double* p = malloc(sizeof(double));
  if (p == NULL)
    exit(0);
  *p = coeff;
  linkedListAppend(polynome, p);
}

/*----------------------------------------------------------*/

/*
  Cree et retourne le polynome de degre n-1 dont les
  coefficients sont passes dans le tableau d.
 */

linkedList* makeArrayPolynome(double* d, int n)
{
  linkedList* l = makePolynome();
  int i;
  for(i = 0 ; i < n ; i++)
    appendCoeffPolynome(l, *(d + i));
  return l;
}

/*----------------------------------------------------------*/

/*
  Retourne le polynome nul.
*/

linkedList* zeroPolynome()
{
  linkedList* l = makePolynome();
  appendCoeffPolynome(l, 0.0);
  return l;
}

/*----------------------------------------------------------*/

/*
  Retourne le polynome unite.
*/

linkedList* unitPolynome()
{
  linkedList* l = makePolynome();
  appendCoeffPolynome(l, 1.0);
  return l;
}

/*----------------------------------------------------------*/

/*
  Retourne le polynome
  0 + X + 2X^2 + 3X^3 + ... + iX^i + ... + nX^n
*/

linkedList* simpleListPolynome(int n)
{
  linkedList* l;
  if (n == 0)
    return zeroPolynome();
  l = simpleListPolynome(n - 1);
  appendCoeffPolynome(l, n);
  return l;
}

/*----------------------------------------------------------*/

/*
  Ajoute la constante d au polynome l.
*/

void addRealPolynome(linkedList* l, double d)
{
  link* lnk = linkedListGetFirst(l);
  if (lnk != NULL)
    *(double*)lnk->data += d;
  else
    appendCoeffPolynome(l, d);
}
```

```
/*-------------------------------------------------------*/

/*
  Multiplie le polynome l par le reel d.
*/

void multRealPolynome(linkedList* l, double d)
{
  void multiplyDouble(void* v)
    {
      *(double*)v *= d;
    }
  linkedListApply(l, multiplyDouble);
}

/*-------------------------------------------------------*/

/*
  Multiplie le polynome l par X^n
*/

void multXNPolynome(linkedList* l, int n)
{
  if (n > 0)
    {
      linkedListPush(l, makeDouble(0.0));
      multXNPolynome(l, n - 1);
    }
}

/*-------------------------------------------------------*/

/*
  Multiplie le polynome l par coeff*X^exp
*/

void multMonomePolynome(linkedList* l, double coeff, int exp)
{
  multRealPolynome(l, coeff);
  multXNPolynome(l, exp);
}

/*-------------------------------------------------------*/

/*
  Additionne deux Ã  deux tous les maillons des deux listes la
  et lb. Concatene le resultat a la liste result.
*/

void addMaillons(link* la, link* lb, linkedList* result)
{
  if (la != NULL && lb != NULL)
    {
      appendCoeffPolynome(result, *(double*)la->data
                          + *(double*)lb->data);
      addMaillons(la->next, lb->next, result);
    }
  else
    {
      if (la != NULL)
        {
          appendCoeffPolynome(result, *(double*)la->data);
          addMaillons(la->next, NULL, result);
        }
      if (lb != NULL)
        {
          appendCoeffPolynome(result, *(double*)lb->data);
          addMaillons(NULL, lb->next, result);
        }
    }
}

/*-------------------------------------------------------*/

/*
  Additione les deux polynomes a et b, retourne cette somme.
*/

linkedList* addPolynome(linkedList* a, linkedList* b)
{
  linkedList* result = makePolynome();
```

```
  addMaillons(linkedListGetFirst(a),
              linkedListGetFirst(b),
              result);
  return result;
}

/*-------------------------------------------------------*/

/*
  Multiplie a et b avec la recurrence
  (a_0 + a_1X + ... + a_nX^n)Q(X)
  = a_0 * Q(X) + X * (a_1 + a_2 X...+ a_n X^(n-1)) * Q(X)
*/

linkedList* slowMultPolynome(linkedList* a, linkedList* b)
{
  linkedList* temp1 = NULL;
  linkedList* res = zeroPolynome();
  int n = 0;
  void auxFunction(void* v)
    {
      linkedList* temp2 = copyPolynome(b);
      multMonomePolynome(temp2, *(double*)v, n);
      temp1 = addPolynome(res, temp2);
      destroyPolynome(res);
      destroyPolynome(temp2);
      res = temp1;
      n++;
    }
  linkedListApply(a, auxFunction);
  return res;
}

/*-------------------------------------------------------*/

/*
  Fonction auxiliaire pour le sous-programme ci-dessous.
*/

void multMaillons(link* la, link* lb, linkedList* result)
{
  pComplexe p;
  if (la != NULL)
    {
      p = copyComplexe(
                       multComplexe(
                                    *(pComplexe)la->data,
                                    *(pComplexe)lb->data));
      linkedListAppend(result, p);
      multMaillons(la->next, lb->next, result);
    }
}


/*-------------------------------------------------------*/
/*
  Multiplie entre eux tous les coefficients (complexes) des monomes de
  meme degre, par exemple :
  (1 + 2X + 4X^2) et (3 + X - 2X^2) donnent
  (1 * 3) + (2 * 1)X + (4 * -2)X^2
  a et b sont necessairement de meme degre.
*/

linkedList* multMonomesPolynome(linkedList* a, linkedList* b)
{
  linkedList* l = makePolynome();
  multMaillons(linkedListGetFirst(a),
               linkedListGetFirst(b),
               l);
  return l;
}

/*-------------------------------------------------------*/

/*
  Convertit un double en pointeur sur complexe.
 */

void* complexOfDouble(void* v)
{
  return copyComplexe(makeAffComplexe(*(double*)v, 0.0));
```

```
}

/*-------------------------------------------------------*/

/*
  Convertit un complexe en double, supprime
  la partie imaginaire.
*/

void* doubleOfComplexe(void* v)
{
  return makeDouble(reComplexe(*(pComplexe)v));
}

/*-------------------------------------------------------*/

/*
  Detruit un nombre complexe.
*/

void destroyComplexe(void* v)
{
  free((pComplexe)v);
}

/*-------------------------------------------------------*/

/*
  Retourne la premier puisance de 2 superieure ou egal a x.
*/

int puiss2UpperThan(int x)
{
  int k = 2;
  while(k < x)
    k *= 2;
  return k;
}

/*-------------------------------------------------------*/

/*
  Ajoute des zeros non significatifs a la fin de l de sorte
  que sa taille devienne k.
 */

void resizeList(linkedList* l, int k)
{
  while(linkedListGetSize(l) < k)
    linkedListAppend(l,
                     copyComplexe(makeAffComplexe(0.0, 0.0)));
}

/*-------------------------------------------------------*/

/*
  Modifie les tailles des deux listes de sorte
  qu'on puisse leur appliquer une FFT
 */

void calibrateLists(linkedList* l1, linkedList* l2)
{
  int k;
  int length = linkedListGetSize(l1) +
    linkedListGetSize(l2) - 1;
  k = puiss2UpperThan(length);
  resizeList(l1, k);
  resizeList(l2, k);
}

/*-------------------------------------------------------*/

/*
  Multiple a et b en passant par une transformee de fourier.
 */

linkedList* multPolynome(linkedList* a, linkedList* b)
{
  linkedList *ca, *cb, *ffta, *fftb, *prodab, *resComplexe, *res;
  ca = linkedListMap(a, complexOfDouble);
  cb = linkedListMap(a, complexOfDouble);
```

```
  calibrateLists(ca, cb);
  ffta = FFT(ca);
  fftb = FFT(cb);
  prodab = multMonomesPolynome(ffta, fftb);
  resComplexe = inverseFFT(prodab);
  res = linkedListMap(resComplexe, doubleOfComplexe);
  linkedListDestroy(ca, destroyComplexe);
  linkedListDestroy(cb, destroyComplexe);
  linkedListDestroy(ffta, destroyComplexe);
  linkedListDestroy(fftb, destroyComplexe);
  linkedListDestroy(prodab, destroyComplexe);
  linkedListDestroy(resComplexe, destroyComplexe);
  return res;
}

/*-------------------------------------------------*/

/*
  Evalue le polynome l en x avec la methode de Horner.
 */

double hornerLink(link* lnk, double x)
{
  if (lnk == NULL)
    return 0.0;
  return *(double*)lnk->data +
    x*hornerLink(lnk->next, x);
}

/*-------------------------------------------------*/

/*
  Evalue le polynome l en x avec la methode de Horner.
 */

double evaluatePolynome(linkedList *l, double x)
{
  return hornerLink(linkedListGetFirst(l), x);
}

/*-------------------------------------------------*/

int main()
{
  linkedList *l = simpleListPolynome(10000);
  //makeArrayPolynome(d, 5);
  linkedList *m, *n, *p;
  printf("multiplication\n");
  m = slowMultPolynome(l, l);
  printf("done\n");
  //printPolynome(m);
  printf("multiplication\n");
  n = multPolynome(l, l);
  printf("done\n");
  //printPolynome(m);
  multRealPolynome(n, -1);
  p = addPolynome(m, n);
  printf("test : %lf\n", evaluatePolynome(p, 1.0));
  destroyPolynome(l);
  destroyPolynome(m );
  destroyPolynome(n );
  return 0;
}
```

## A.14 Tas binomiaux

Voici une implémentation des tas binomiaux, j'ai commencé par écrire quelques fonctions de gestion de listes chaînées, et j'ai crée deux classes pour représenter respectivement les arbres binomiaux et les tas binomiaux.

```
/**
   Implements a very simple linked list, each link points to item
   of type T and references the next link. O(1) operations are
   getting the first link's data or the second's link reference,
   or deleting the first element. Each functions modifying the
   linking returns a pointer to the first link of the list.

 */

public class LinkedList<T>
{
    /*********************************************/

    private T data;

    /*********************************************/

    private LinkedList<T> next;

    /*********************************************/

    /**
       Creates a one element list.
     */

    public LinkedList(T data)
    {
        this.data = data;
    }

    /*********************************************/

    /**
       Creates a one link list and appends the second
       paramter to this list.
     */

    public LinkedList(T data, LinkedList<T> next)
    {
        this(data);
        this.next = next;
    }

    /*********************************************/

    /**
       Modifies the data.
     */

    public void setData(T data)
    {
        this.data = data;
    }

    /*********************************************/

    /**
       Appends a list to the first link of the current
       list.
     */

    public void setNext(LinkedList<T> next)
    {
        this.next = next;
    }

    /*********************************************/

    /**
       Retrives the data of the first link.
     */

    public T getData()
    {
        return data;
    }
```

```
/*********************************************/

/**
   Returns the second item of the list, doesn't change
   the linking
 */
public LinkedList<T> getNext()
{
    return next;
}

/*********************************************/

/**
   Delete from this list the link which the adress is l,
   returns the list whithout l.
 */
public LinkedList<T> deleteLink(LinkedList<T> l)
{
    if (this == l)
        return getNext();
    setNext(getNext().deleteLink(l));
    return this;
}

/*********************************************/

/**
   Returns a String representation of the list.
 */
public String toString()
{
    return " -> " + data +
        ((next != null) ? next.toString() : "X");
}

/*********************************************/

private static <K> LinkedList<K> reverseAcc(LinkedList<K> link,
                                            LinkedList<K> acc)
{
    if (link == null)
        return acc;
    LinkedList<K> next = link.getNext();
    link.setNext(acc);
    return reverseAcc(next, link);
}

/*********************************************/

/**
   Adds the link l in front of this list, returns
   the list thus created.
 */
public LinkedList<T> addFront(LinkedList<T> l)
{
    l.setNext(this);
    return l;
}

/*********************************************/

/**
   Adds the item o in front of this list, returns
   the new list.
 */
public LinkedList<T> addFront(T o)
{
    return addFront(new LinkedList<T>(o));
}

/*********************************************/

/**
   Reverse the order of the items of the list. Returns the
```

```
        new list.
     */

    public LinkedList<T> reverse()
    {
        return reverseAcc(this, null);
    }

}

class BinomialTree<T extends Comparable<T>>
    implements Comparable<BinomialTree<T>>
{

    /**********************************************/

    private int order;

    /**********************************************/

    private T data;

    /**********************************************/

    private LinkedList<BinomialTree<T>> sons;

    /**********************************************/

    public BinomialTree(T data)
    {
        this.data = data;
        order = 0;
        sons = null;
    }

    /**********************************************/

    public BinomialTree<T> fusion(BinomialTree<T> other)
    {
        if (getData().compareTo(other.getData()) > 0)
            return other.fusion(this);
        if (order != other.getOrder())
            System.out.println("Invalid arguments");
        sons = (sons == null) ?
            new LinkedList<BinomialTree<T>>(other) :
            sons.addFront(other);
        order++;
        return this;
    }

    /**********************************************/

    public int getOrder()
    {
        return order;
    }

    /**********************************************/

    public T getData()
    {
        return data;
    }

    /**********************************************/

    public BinomialHeap<T> extractSons()
    {
        return new BinomialHeap<T>((sons == null) ?
                                   null :
                                   sons.reverse()
                                   );
    }

    /**********************************************/

    public int compareTo(BinomialTree<T> other)
    {
        if (order < other.getOrder())
            return -1;
        if (order == other.getOrder())
```

```
            return 0;
        return 1;
    }
}

/**
  Implements a priority queue which allows the following operations,
  each one in a in O(log n) time.
  - inserting an item
  - getting the minimum key item
  - deleting the minimum key item
  - merging two queues
  It has been implemented by alexandre-mesle.com for java 1.5
*/

public class BinomialHeap<T extends Comparable<T>>
{

    /**********************************************/

    private LinkedList<BinomialTree<T>> binomialTrees;

    /**********************************************/

    /**
       Creates an empty queue.
     */

    public BinomialHeap()
    {
        binomialTrees = null;
    }

    /**********************************************/

    /**
       Creates a one item queue.
     */

    public BinomialHeap(T data)
    {
        binomialTrees = new LinkedList<BinomialTree<T>>
            (new BinomialTree<T>(data));
    }

    /**********************************************/

    BinomialHeap(LinkedList<BinomialTree<T>> l)
    {
        binomialTrees = l;
    }

    /**********************************************/

    /**
       Returns trus if and only if the queue doesn't contain
       any item.
     */

    public boolean isEmpty()
    {
        return binomialTrees == null;
    }

    /**********************************************/

    private static <K extends Comparable<K>> LinkedList<BinomialTree<K>>
        findMinLink(LinkedList<BinomialTree<K>> link,
                    LinkedList<BinomialTree<K>> acc)
    {
        if (link == null)
            return acc;
        K linkData = link.getData().getData();
        K accData = acc.getData().getData();
        if (linkData.compareTo(accData) < 1)
            return findMinLink(link.getNext(), link);
        return findMinLink(link.getNext(), acc);
    }

    /**********************************************/
```

```
/**
   Returns the item wich has the minimum key.
 */

public T getMin()
{
    LinkedList<BinomialTree<T>> minNode = findMinLink(binomialTrees,
                                                      binomialTrees);
    if (minNode == null)
        return null;
    return minNode.getData().getData();
}


/*********************************************/

/**
   Deletes the minimum key item.
 */

public void deleteMin()
{
    LinkedList<BinomialTree<T>> minLink =
        findMinLink(binomialTrees, binomialTrees);
    binomialTrees = binomialTrees.deleteLink(minLink);
    BinomialHeap<T> sons = minLink.getData().extractSons();
    fusion(sons);
}

/*********************************************/

/**
   Deletes the minimum key item.
 */

public void insert(T c)
{
    fusion(new BinomialHeap<T>(c));
}

/*********************************************/

/**
   Adds to the current queue all the item of the queue b.
   Use it only if b doesn't have to used again.
 */

public void fusion(BinomialHeap<T> b)
{
    binomialTrees =
        clean(merge(binomialTrees, b.binomialTrees));
}

/*********************************************/

private static <K extends Comparable<K>> LinkedList<BinomialTree<K>>
    merge(LinkedList<BinomialTree<K>> l1,
          LinkedList<BinomialTree<K>> l2)
{
    if (l1 == null)
        return l2;
    if (l2 == null)
        return l1;
    if (l1.getData().compareTo(l2.getData()) <= 0)
        {
            l1.setNext(merge(l1.getNext(), l2));
            return l1;
        }
    else
        return merge(l2, l1);
}

/*********************************************/

private static <K extends Comparable<K>> LinkedList<BinomialTree<K>>
    clean(LinkedList<BinomialTree<K>> l)
{
    if (l == null)
        return null;
    if (l.getNext() == null)
        return l;
```

```
        BinomialTree<K> first = l.getData();
        BinomialTree<K> second = l.getNext().getData();
        if (first.getOrder() != second.getOrder())
            {
                l.setNext(clean(l.getNext()));
                return l;
            }
        if (l.getNext().getNext() == null)
            return new LinkedList<BinomialTree<K>>
                (first.fusion(second),
                 l.getNext().getNext());
        BinomialTree<K> third = l.getNext().getNext().getData();
        if (first.getOrder() == third.getOrder())
            {
                l.setNext(clean(l.getNext()));
                return l;
            }
        else
            return new LinkedList<BinomialTree<K>>
                (first.fusion(second),
                 l.getNext().getNext());
    }

    /*********************************************/

    /*

    public static void main(String[] args)
    {
        BinomialHeap<Integer> h = new BinomialHeap<Integer>();
        for (int i = 1 ; i < 3000 ; i++)
            h.insert(new Integer((i*i*i*i*i*i*i) % 127));
        while(!h.isEmpty())
            {
                System.out.print(h.getMin() + " -> ");
                h.deleteMin();
            }
    }

    */

}
```

## A.15 AVL

```c
#include<stdio.h>
#include<malloc.h>
#include"avl.h"

/*--------------------------------------------------------------*/

/*
  Noeud de l'ABR.
 */

typedef struct nd
{
  /*
    key est la cle du noeud courant.
   */
  int key;

  /*
    pointeur vers l'element de cle key
   */
  void* data;

  /*
    hauteur du sous-arbre :
    0 si ce noeud est une feuille.
   */
  int height;

  /*
    Pointeur vers le sous-arbre droit.
   */
  struct nd * left;

  /*
    Pointeur vers le sous-arbre gauche.
   */
  struct nd * right;
}node;

/*--------------------------------------------------------------*/

/*
  Retourne la hauteur de l'arbre de racine l,
  -1 si l est vide.
 */

int getHeight(node* l)
{
  if (l == NULL)
    return -1;
  return l->height;
}

/*--------------------------------------------------------------*/

/*
  Retourne la plus grande des deux valeurs i et j.
 */

int max(int i, int j)
{
  if (i < j)
    return j;
  return i;
}

/*--------------------------------------------------------------*/

/*
  Met a jour la hauteur de la racine l en fonction des
  hauteurs des racines des deux sous-arbres.
 */

void setHeight(node* l)
{
  if (l != NULL)
    {
        l->height = 1 + max(getHeight(l->left), getHeight(l->right));
    }
```

```
}

/*-------------------------------------------------------*/

/*
  Cree un noeud contenant la donnee data,
  ayant pour sous-arbre gauche l et
  pour sous-arbre droit r.
*/

node* nodeCreate(int (*getKey)(void*), node* l, void* data, node* r)
{
  node* n = (node*)malloc(sizeof(node));
  if (n == NULL)
    error();
  n->data = data;
  n->left = l;
  n->right = r;
  n->key = getKey(data);
  setHeight(n);
  return n;
}

/*-------------------------------------------------------*/

/*
  Retourne un avl vide
*/

avl* avlCreate(int (*getKey)(void*), void (*freeData)(void*))
{
  avl* a = (avl*)malloc(sizeof(avl));
  if (!a)
    error();
  a->getKey = getKey;
  a->freeData = freeData;
  a->root = NULL;
  return a;
}

/*-------------------------------------------------------*/

/*
  Fait de freeData la fonction de destruction des donnees.
*/

void avlSetFreeFunction(avl* a, void (*freeData)(void*))
{
  a->freeData = freeData;
}

/*-------------------------------------------------------*/

/*
  Affiche toutes les cles du sous-arbre de racine n dans
  l'ordre croissant.
*/

void nodePrintKeys(node* n)
{
  if(n)
    {
      printf("(");
      nodePrintKeys(n->left);
      printf(", %d, ", n->key );
      nodePrintKeys(n->right);
      printf(")");
    }
}

/*-------------------------------------------------------*/

/*
  Affiche pour tous les noeuds du sous-arbre de racine n
  la difference entre les hauteurs du sous-arbre droit et
  gauche.
*/

void nodePrintLevels(node* n)
{
  if(n)
```

```
    {
      printf("(");
      nodePrintLevels(n->left);
      printf(", %d, ", getHeight(n->left)- getHeight(n->right));
      nodePrintLevels(n->right);
      printf(")");
    }
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation droite de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateRight(node* x)
{
  node* y = x->left;
  node* b = y->right;
  y->right = x;
  x->left = b;
  setHeight(x);
  setHeight(y);
  return y;
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation gauche de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateLeft(node* x)
{
  node* y = x->right;
  node* b = y->left;
  y->left = x;
  x->right = b;
  setHeight(x);
  setHeight(y);
  return y;
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation gauche-droite de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateLeftRight(node* x)
{
  x->left = rotateLeft(x->left);
  return rotateRight(x);
}

/*-------------------------------------------------------*/

/*
  Effectue une rotation droite-gauche de l'arbre de racine x,
  retourne la racine de l'arbre apres rotation.
 */

node* rotateRightLeft(node* x)
{
  x->right = rotateRight(x->right);
  return rotateLeft(x);
}

/*-------------------------------------------------------*/

/*
  Reequilibre l'arbre de racine x, retourne la racine de
  l'arbre apres reequilibrage. On part du principe
  que les hauteurs des sous-arbres droit et gauche different
  de au plus 1.
*/

node* balance(node* x)
```

```
{
  if (x != NULL)
    {
      setHeight(x);
      if (getHeight(x->left) + 2 == getHeight(x->right))
        {
          if(getHeight(x->left) + 1 == getHeight(x->right->right))
            return rotateLeft(x);
          else
            return rotateRightLeft(x);
        }
      if (getHeight(x->left) == getHeight(x->right) + 2)
        {
          if(getHeight(x->left->left) == getHeight(x->right) + 1)
            return rotateRight(x);
          else
            return rotateLeftRight(x);
        }
    }
  return x;
}

/*-------------------------------------------------------------*/

/*
  Insere la donnee v dans l'arbre de racine n
  et reequilibre l'arbre, retourne la racine de
  l'arbre apres insertion et reequilibrage.
 */

node* nodeInsert(int (*key)(void*), node* n, void* v)
{
  if(n == NULL)
    return nodeCreate(key, NULL, v, NULL);
  if(key(v) == n->key)
    return n;
  if(key(v) < n->key)
    n->left = nodeInsert(key, n->left, v);
  else
    n->right = nodeInsert(key, n->right, v);
  return balance(n);
}

/*-------------------------------------------------------------*/

/*
  Insere la donnee v dans l'arbre a, O(log n).
*/

void avlInsert(avl* a, void* v)
{
  a->root = nodeInsert(a->getKey, (node*)a->root, v);
}

/*-------------------------------------------------------------*/

/*
  Affiche les cles de l'AVL a dans l'ordre croissant, O(n)
*/

void avlPrintKeys(avl* a)
{
  nodePrintKeys((node*)a->root);
  printf("\n");
}

/*-------------------------------------------------------------*/

/*
  Affiche pour chaque noeud la difference entre les hauteurs
  des sous-arbres droit et gauche de l'AVL.
*/

void avlPrintLevels(avl* a)
{
  nodePrintLevels((node*)a->root);
  printf("\n");
}

/*-------------------------------------------------------------*/
```

```
/*
  Retourne la donnee de cle k si elle se trouve dans le sous-arbre
  de racine n, NULL sinon.
*/

void* findNode(node* n, int k)
{
  if (n == NULL)
    return NULL;
  if (n->key == k)
    return n->data;
  if (k < n->key)
    return findNode(n->left, k);
  else
    return findNode(n->right, k);
}

/*-------------------------------------------------------*/

/*
  Retourne la donnee de cle x si elle se trouve dans l'AVL a,
  NULL sinon, O(log n)
*/

void* avlFind(avl* a, int x)
{
  return findNode((node*)a->root, x);
}

/*-------------------------------------------------------*/

/*
  Fait pointer *max vers le noeud de cle maximale dans
  le sous-arbre de racine n, detache le noeud *max de l'arbre
  et retourne la racine de l'arbre apres suppression.
*/

node* removeMax(node* n, node** max)
{
  if (n == NULL)
    {
      *max = NULL;
      return NULL;
    }
  if (n->right == NULL)
    {
      *max = n;
      return NULL;
    }
  n->right = removeMax(n->right, max);
  return balance(n);
}

/*-------------------------------------------------------*/

/*
  Fait pointer *min vers le noeud de cle minimale dans
  le sous-arbre de racine n, detache le noeud *min de l'arbre
  et retourne la racine de l'arbre apres suppression.
*/

node* removeMin(node* n, node** min)
{
 if (n == NULL)
    {
      *min = NULL;
      return NULL;
    }
  if (n->left == NULL)
    {
      *min = n;
      return NULL;
    }
  n->left = removeMin(n->left, min);
  return balance(n);
}

/*-------------------------------------------------------*/

/*
  Supprime le noeud de cle k du sous-arbre de racine n,
```

```
  applique la fonction de destruction a la donnee de ce noeud,
  et retourne la racine de l'arbre apres la suppression.
*/

node* removeNode(node* n, int k, void (*freeData)(void*))
{
  node* p;
  if (n == NULL)
    return NULL;
  if (k == n->key)
    {
      p = NULL;
      n->left = removeMax(n->left, &p);
      if (p == NULL)
        n->right = removeMin(n->right, &p);
      if (p != NULL)
        {
          p->left = n->left;
          p->right = n->right;
        }
      freeData(n->data);
      free(n);
      return balance(p);
    }
  if (k < n->key)
    n->left = removeNode(n->left, k, freeData);
  else
    n->right = removeNode(n->right, k, freeData);
  return balance(n);
}

/*-------------------------------------------------------------*/

/*
  Supprime le noeud de cle k de l'AVL a, applique la fonction
  de destruction a la donnee de cle k, O(log n).
*/

void avlRemove(avl* a, int k)
{
  a->root = removeNode((node*)a->root, k, a->freeData);
}

/*-------------------------------------------------------------*/

/*
  Place dans la liste chainee l toutes les donnees
  du sous-arbre de racine n dans l'ordre croissant.
 */

void nodeToList(node* n, linkedList* l)
{
  if (n != NULL)
    {
      nodeToList(n->left, l);
      linkedListAppend(l, n->data);
      nodeToList(n->right, l);
    }
}

/*-------------------------------------------------------------*/

/*
  Retourne une liste chainee contenant toutes les donnees
  de l'AVL a disposees dans l'ordre croissant, O(n).
 */

linkedList* avlToList(avl* a)
{
  linkedList* l = linkedListCreate();
  nodeToList((node*)a->root, l);
  return l;
}

/*-------------------------------------------------------------*/

/*
  Detruit tous les noeuds du sous-arbre de racine n,
  applique la fonction de destruction fr a toutes les
  donnees du sous-arbre.
*/
```

```
void nodeDestroy(node* n, void (*fr)(void*))
{
  if (n != NULL)
    {
      nodeDestroy(n->left, fr);
      nodeDestroy(n->right, fr);
      fr(n->data);
      free(n);
    }
}

/*-------------------------------------------------------*/

/*
  Detruit l'AVL a, applique la fonction de destruction a
  toutes les donnees du sous-arbre, O(n).
*/

void avlDestroy(avl* a)
{
  nodeDestroy((node*)a->root, a->freeData);
  free(a);
}

/*-------------------------------------------------------*/
/*-------------------------------------------------------*/

/*
Pour tester les algorithmes
*/

void destroyInt(void* i)
{
  free((int*)i);
}

int getIntKey(void* i)
{
  return *((int*)i);
}

void doNothing()
{

}

int main()
{
  int i;
  int* d;
  linkedList* l;
  link* m;
  avl* a = avlCreate(getIntKey, destroyInt);
  for (i = 0 ; i < 40 ; i++)
    {
      d = (int*)malloc(sizeof(int));
      *d = i;
      avlInsert(a, d);
    }
  avlPrintKeys(a);
  avlPrintLevels(a);
  for(i = 0 ; i < 40 ; i+=5)
    avlRemove(a, i);
  avlPrintKeys(a);
  avlPrintLevels(a);
  l = avlToList(a);
  for(m = linkedListGetFirst(l); m != NULL ; m = m->next)
    printf("%d -> ", *(int*)(m->data));
  printf("\n");
  linkedListDestroy(l, doNothing);
  avlSetFreeFunction(a, destroyInt);
  avlDestroy(a);
  return 0;
}
```

## A.16 Knapsack par recherche exhaustive

```
#include"linkedList.h"
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>

/*-------------------------------------------------------------*/

/*
  Instance d'un probleme de knapSack
*/

typedef struct
{
  /*
    nombre total d'objets
   */
  long nbItems;

  /*
    tableau de nombreObjets elements contenant le poids de
    chaque element
  */
  long* weights;

  /*
    tableau de nombreObjets elements contenant la valeur de
    chaque element
  */
  long* values;

  /*
    Poids maximum qu'il est possible de placer dans le sac Ã  dos.
   */
  long maxWeight;
}knapSack;

/*-------------------------------------------------------------*/

knapSack* knapSackMake(long nbItems, long maxWeight)
{
  knapSack* k = (knapSack*)malloc(sizeof(knapSack));
  k->nbItems = nbItems;
  k->maxWeight = maxWeight;
  k->weights = (long*)malloc(sizeof(long)*nbItems);
  if (k->weights == NULL)
    exit(0);
  k->values = (long*)malloc(sizeof(long)*nbItems);
  if (k->values == NULL)
    exit(0);
  return k;
}

/*-------------------------------------------------------------*/

void knapSackSetItem(knapSack* k, long index, long value, long weight)
{
  *(k->weights + index) = weight;
  *(k->values + index) = value;
}

/*-------------------------------------------------------------*/

long knapSackValue(knapSack* instance, linkedList* solution)
{
  long res = 0;
  void incrValue(void* i)
    {
      res += *(instance->values  + *((long*)i));
    }
  linkedListApply(solution, incrValue);
  return res;
}

/*-------------------------------------------------------------*/

long knapSackWeight(knapSack* instance, linkedList* solution)
{
  long res = 0;
```

```
  void incrValue(void* i)
    {
      res += *(instance->weights  + *((long*)i));
    }
  linkedListApply(solution, incrValue);
  return res;
}

/*--------------------------------------------------------------------*/

int knapSackIsFeasible(knapSack* instance, linkedList* solution)
{
  return knapSackWeight(instance, solution)
    <= instance->maxWeight;
}

/*--------------------------------------------------------------------*/

void knapSackPrintInstance(knapSack* k)
{
  long l = 0;
  while(l < k->nbItems)
    {
      printf("(i = %3ld, v = %3ld, w = %3ld)\n", l,
             *(k->values + l), *(k->weights + l));
      l++;
    }
  printf("max weight = %ld\n", k->maxWeight);
}

/*--------------------------------------------------------------------*/

void knapSackDestroy(knapSack* k)
{
  free(k->values);
  free(k->weights);
  free(k);
}

/*--------------------------------------------------------------------*/

void* linkedListCopy(void* l)
{
  void* id(void* i)
    {
      return i;
    }
  return linkedListMap((linkedList*)l, id);
}

/*--------------------------------------------------------------------*/

void knapSackPrintSolution(linkedList* l)
{
  void printIndex(void* i)
    {
      printf("%ld ", *((long*)i));
    }
  linkedListApply(l, printIndex);
}

/*--------------------------------------------------------------------*/

linkedList* knapSackListInit(long n)
{
  linkedList* l;
  if (n <= 0)
    l = linkedListCreate();
  else
    {
      l = knapSackListInit(n - 1);
      long* p = malloc(sizeof(long));
      if (p == NULL)
        exit(0);
      *p = n-1;
      linkedListAppend(l, p);
    }
  return l;
}

/*--------------------------------------------------------------------*/
```

```
void knapSackPrintParties(linkedList* l)
{
  void knapSackPrintVoidSolution(void* l)
    {
      printf("[ ");
      knapSackPrintSolution((linkedList*)l);
      printf("]\n");
    }
  linkedListApply(l, knapSackPrintVoidSolution);
}

/*--------------------------------------------------------------*/

void doNothing(void* v)
{
}

/*--------------------------------------------------------------*/

void destroyInt(void* v)
{
  free((int*)v);
}

/*--------------------------------------------------------------*/

void knapSackDestroyParties(linkedList* l)
{
  void destroyPartie(void* p)
    {
      linkedListDestroy((linkedList*)p, doNothing);
    }
  linkedListDestroy(l, destroyPartie);
}

/*--------------------------------------------------------------*/

linkedList* parties(linkedList* l)
{
  linkedList* res = linkedListCreate();
  linkedList *partiesWithoutX, *partiesWithX;
  if (linkedListGetSize(l) == 0)
    {
      linkedListAppend(res, linkedListCreate());
    }
  else
    {
      link* x = linkedListUnlinkFirst(l);
      partiesWithoutX = parties(l);
      partiesWithX = linkedListMap(partiesWithoutX, linkedListCopy);
      void appendX(void* data)
        {
          linkedListAppend((linkedList*)data, x->data);
        }
      linkedListApply(partiesWithX, appendX);
      linkedListAppendLink(l, x);
      linkedListConcat(res, partiesWithoutX);
      linkedListConcat(res, partiesWithX);
      linkedListDestroy(partiesWithX, doNothing);
      linkedListDestroy(partiesWithoutX, doNothing);
    }
  printf("-> 2^%d\n", linkedListGetSize(l));
  return res;
}

/*--------------------------------------------------------------*/

linkedList* knapSackBruteForceSolve(knapSack* k)
{
  linkedList* indexes = knapSackListInit(k->nbItems);
  linkedList* allSolutions = parties(indexes);
  linkedList* bestSolutionSeen = NULL;
  linkedListGetFirst(allSolutions);
  printf("-> subsets generated.\nTrying to find the best solution\n");
  void findBest(void* v)
    {
      linkedList* l = (linkedList*)v;
      if (knapSackIsFeasible(k, l)
          && (bestSolutionSeen == NULL ||
              knapSackValue(k, l) > knapSackValue(k, bestSolutionSeen)))
```

```
        bestSolutionSeen = l;
      }
    linkedListApply(allSolutions, findBest);
    printf(" -> Done.\nDestroying the subsets\n");
    void* copyInt(void* i)
      {
        void* v = malloc(sizeof(long));
        if (v == NULL)
          exit(0);
        *((long*)v) = *((long*)i);
        return v;
      }
    bestSolutionSeen = linkedListMap(bestSolutionSeen, copyInt);
    void destroyTheLinkedList(void* v)
      {
        linkedListDestroy((linkedList*)v, destroyInt);
      }
    linkedListDestroy(allSolutions, destroyTheLinkedList);
    printf(" -> Done.\nThanks for your patience.\n");
    return bestSolutionSeen;
}

/*--------------------------------------------------------------------*/

int main()
{
    long t = 21;
    knapSack* k = knapSackMake(t, 40);
    linkedList* s;
    long l;
    for(l = 0 ; l < k->nbItems ; l++)
      knapSackSetItem(k, l, (l*l*l)%10 + 1, (l*l*l*l)%10 + 1);
    knapSackPrintInstance(k);
    s = knapSackBruteForceSolve(k);
    knapSackPrintSolution(s);
    printf("\n");
    linkedListDestroy(s, doNothing);
    knapSackDestroy(k);
    return 0;
}
```

## A.17 Prise en main de GLPK

```c
#include<stdio.h>
#include"glpk.h"

int main()
{
  LPX* lp = lpx_create_prob();
  double coeffs[10];
  int cols[10], rows[10];
  int k = 1;
  lpx_set_obj_dir(lp, LPX_MAX);
  lpx_add_cols(lp, 3);
  lpx_set_obj_coef(lp, 1, 4);
  lpx_set_obj_coef(lp, 2, 6);
  lpx_set_obj_coef(lp, 3, 2);
  lpx_set_col_bnds(lp, 1, LPX_LO, 0.0, 0.0);
  lpx_set_col_bnds(lp, 2, LPX_LO, 0.0, 0.0);
  lpx_set_col_bnds(lp, 3, LPX_LO, 0.0, 0.0);
  lpx_add_rows(lp, 3);
  lpx_set_row_bnds(lp, 1, LPX_UP, 0.0, 12.);
  lpx_set_row_bnds(lp, 2, LPX_UP, 0.0, 3.);
  lpx_set_row_bnds(lp, 3, LPX_UP, 0.0, 6.);
  coeffs[k] = 3.; rows[k] = 1; cols[k] = 1; k++;
  coeffs[k] = 7.; rows[k] = 1; cols[k] = 2; k++;
  coeffs[k] = 1.; rows[k] = 1; cols[k] = 3; k++;
  coeffs[k] = 0.6; rows[k] = 2; cols[k] = 1; k++;
  coeffs[k] = 3; rows[k] = 2; cols[k] = 2; k++;
  coeffs[k] = 2.8; rows[k] = 2; cols[k] = 3; k++;
  coeffs[k] = 9.; rows[k] = 3; cols[k] = 1; k++;
  coeffs[k] = 0.08; rows[k] = 3; cols[k] = 2; k++;
  coeffs[k] = 5.; rows[k] = 3; cols[k] = 3; k++;
  lpx_load_matrix(lp, 9, rows, cols, coeffs);
  lpx_simplex(lp);
  printf("fonction obectif = %lf\n", lpx_get_obj_val(lp));
  for(k = 1 ; k <= 3 ; k++)
    printf("x_%d = %lf\n", k, lpx_get_col_prim(lp, k));
  lpx_delete_prob(lp);
  return 0;
}
```

## A.18 Le plus beau métier du monde

```
#include<stdio.h>
#include"glpk.h"

#define N 10

double sommeTab(double* t, int n)
{
  if (n == 0)
    return 0.;
  return *t + sommeTab(t+1, n-1);
}

void question1(double *notesPartiel, double* notesCC1, double* notesCC2)
{
  double coeffs[4];
  int rows[4], cols[4];
  LPX* lp;
  int k = 1;
  lp = lpx_create_prob();
  lpx_set_obj_dir(lp, LPX_MAX);
  lpx_add_cols(lp, 3);
  lpx_set_obj_coef(lp, 1, sommeTab(notesPartiel, N)/N);
  lpx_set_obj_coef(lp, 2, sommeTab(notesCC1, N)/N);
  lpx_set_obj_coef(lp, 3, sommeTab(notesCC2, N)/N);
  for(k = 1 ; k <= 3 ; k++)
    lpx_set_col_bnds(lp, k, LPX_DB, 0.0, 1.0);
  lpx_add_rows(lp, 1);
  lpx_set_row_bnds(lp, 1, LPX_UP, 0.0, 1.0);
  for(k = 1 ; k <= 3 ; k++)
    {
      rows[k]=1; cols[k]=k; coeffs[k] = 1.0 ;
    }
  lpx_load_matrix(lp, 3, rows, cols, coeffs);
  lpx_simplex(lp);
  lpx_get_obj_val(lp);
  for(k = 1 ; k <= 3 ; k++)
    printf("coeff %d is %lf\n", k, lpx_get_col_prim(lp, k));
  lpx_delete_prob(lp);
}

void question2(double *notesPartiel, double* notesCC1, double* notesCC2)
{
  double coeffs[4];
  int rows[4], cols[4];
  LPX* lp;
  int k = 1;
  lp = lpx_create_prob();
  lpx_set_obj_dir(lp, LPX_MAX);
  lpx_add_cols(lp, 3);
  lpx_set_obj_coef(lp, 1, sommeTab(notesPartiel, N)/N);
  lpx_set_obj_coef(lp, 2, sommeTab(notesCC1, N)/N);
  lpx_set_obj_coef(lp, 3, sommeTab(notesCC2, N)/N);
  lpx_set_col_bnds(lp, 1, LPX_DB, 0.5, 1.0);
  for(k = 2 ; k <= 3 ; k++)
    lpx_set_col_bnds(lp, k, LPX_DB, 0.1, 1.0);
  lpx_add_rows(lp, 1);
  lpx_set_row_bnds(lp, 1, LPX_UP, 0.0, 1.0);
  for(k = 1 ; k <= 3 ; k++)
    {
      rows[k]=1; cols[k]=k; coeffs[k] = 1.0 ;
    }
  lpx_load_matrix(lp, 3, rows, cols, coeffs);
  lpx_simplex(lp);
  lpx_get_obj_val(lp);
  for(k = 1 ; k <= 3 ; k++)
    printf("coeff %d is %lf\n", k, lpx_get_col_prim(lp, k));
  lpx_delete_prob(lp);
}

void question3(double *notesPartiel, double* notesCC1, double* notesCC2)
{
  double coeffs[4*(N+1) + 1];
  int rows[4*(N+1) + 1], cols[4*(N+1) + 1];
  LPX* lp;
  int eleveIndex, coeffIndex = 1, i;
  lp = lpx_create_prob();
  lpx_set_obj_dir(lp, LPX_MAX);
  lpx_add_cols(lp, 4);
  lpx_set_obj_coef(lp, 1, 1.);
```

```
  lpx_set_col_bnds(lp, 1, LPX_FR, 0.0, 0.0);
  lpx_set_col_bnds(lp, 2, LPX_DB, 0.5, 1.);
  lpx_set_col_bnds(lp, 3, LPX_DB, 0.1, 1.);
  lpx_set_col_bnds(lp, 4, LPX_DB, 0.1, 1.);
  lpx_add_rows(lp, N+1);
  for(eleveIndex = 0 ; eleveIndex < N ; eleveIndex++)
    {
      rows[coeffIndex]=eleveIndex + 1;
      cols[coeffIndex]=1;
      coeffs[coeffIndex] = 1.0 ;
      coeffIndex++;
      rows[coeffIndex]=eleveIndex + 1;
      cols[coeffIndex]=2;
      coeffs[coeffIndex] = -*(notesPartiel + eleveIndex) ;
      coeffIndex++;
      rows[coeffIndex]=eleveIndex + 1;
      cols[coeffIndex]=3;
      coeffs[coeffIndex] = -*(notesCC1 + eleveIndex) ;
      coeffIndex++;
      rows[coeffIndex]=eleveIndex + 1;
      cols[coeffIndex]=4;
      coeffs[coeffIndex] = -*(notesCC2 + eleveIndex) ;
      coeffIndex++;
      lpx_set_row_bnds(lp, eleveIndex + 1, LPX_UP, 0.0, 0.0);
    }
  for(i = 2; i <= 4; i++)
    {
      rows[coeffIndex] = N+1 ;
      cols[coeffIndex] = i;
      coeffs[coeffIndex] = 1.0 ;
      coeffIndex++;
    }
  lpx_set_row_bnds(lp, N+1, LPX_DB, 0.0, 1.0);
  lpx_load_matrix(lp, coeffIndex - 1, rows, cols, coeffs);
  lpx_simplex(lp);
  lpx_get_obj_val(lp);
  for(i = 2 ; i <= 4 ; i++)
    printf("coeff %d is %lf\n", i-1, lpx_get_col_prim(lp, i));
  lpx_print_prob(lp, "question3.lpx");
  lpx_delete_prob(lp);
}

int main()
{
  double notesPartiel[N] = {10, 13,  2, 4.5,  9, 12, 16,  8, 5, 12};
  double notesCC1[N] =     {11, 16, 11,   8, 10, 11, 11,  6, 4, 13};
  double notesCC2[N] =     { 9, 18,  7,   8, 11, 3,  13, 10, 7, 19};
  printf("QUESTION 1\n----------\n");
  question1(notesPartiel, notesCC1, notesCC2);
  printf("QUESTION 2\n----------\n");
  question2(notesPartiel, notesCC1, notesCC2);
  printf("QUESTION 3\n----------\n");
  question3(notesPartiel, notesCC1, notesCC2);
 return 0;
}
```

## A.19 Prise en main de GMP

```
#include<stdio.h>
#include<gmp.h>

/*
  Affiche la table de multiplication de n, de 1 a to.
  n doit etre initialise.
 */

void afficheTableMultiplication(mpz_t n, long to)
{
  mpz_t res;
  if(to > 0)
    {
      afficheTableMultiplication(n, to - 1);
      mpz_init(res);
      mpz_mul_ui(res, n, to);
      mpz_out_str(NULL, 10, res);
      printf("\n");
    }
}


/*
  Place b^n dans res. res doit etre initialise
 */

void puissance(mpz_t b, mpz_t n, mpz_t res)
{
  mpz_t temp;
  if (!mpz_cmp_ui(n, 0))
    {
      mpz_set_ui(res, 1);
    }
  else
    {
      mpz_init(temp);
      if (mpz_divisible_ui_p(n, 2))
        {
          mpz_t q;
          mpz_t deux;
          mpz_mul(temp, b, b);
          mpz_init(q);
          mpz_init_set_ui(deux, 2);
          mpz_cdiv_q(q, n, deux);
          puissance(temp, q, res);
          mpz_clear(q);
          mpz_clear(deux);
        }
      else
        {
          mpz_t nMoinsUn;
          mpz_init(nMoinsUn);
          mpz_sub_ui(nMoinsUn, n, 1);
          puissance(b, nMoinsUn, temp);
          mpz_mul(res, temp, b);
          mpz_clear(nMoinsUn);
        }
      mpz_clear(temp);
    }
}

/*
  Fonction d'Ackermann. res doit etre initialise.
  Attention ! Ne doit etre utilise qu'avec des petites valeurs !
*/

void ackermann(mpz_t m, mpz_t n, mpz_t res)
{
  if(!mpz_cmp_ui(m, 0))
    {
      mpz_add_ui(res, n, 1);
    }
  else
    {
      mpz_t mMoinsUn;
      mpz_init(mMoinsUn);
      mpz_sub_ui(mMoinsUn, m, 1);
      if(!mpz_cmp_ui(n, 0))
        {
```

```
            mpz_t un;
            mpz_init_set_ui(un, 1);
            ackermann(mMoinsUn, un, res);
            mpz_clear(un);
          }
        else
          {
            mpz_t nMoinsUn;
            mpz_t appelRecursif;
            mpz_init(nMoinsUn);
            mpz_init(appelRecursif);
            mpz_sub_ui(nMoinsUn, n, 1);
            ackermann(m, nMoinsUn, appelRecursif);
            ackermann(mMoinsUn, appelRecursif, res);
            mpz_clear(appelRecursif);
            mpz_clear(nMoinsUn);
          }
        mpz_clear(mMoinsUn);
      }
}

/*
  Affiche toutes les images par la fonction d'Ackermann des couples
  {(i, j) | 0 <= i <= m, 0 <= j <= n}
*/

void afficheAckermann(long m, long n)
{
  mpz_t i;
  mpz_t j;
  mpz_t a;
  mpz_init(i);
  mpz_init(j);
  mpz_init(a);
  mpz_set_ui(i, 0);
  while(mpz_cmp_ui(i, m) <= 0)
    {
      mpz_set_ui(j, 0);
      while(mpz_cmp_ui(j, n) <= 0)
        {
          printf("ackermann(");
          mpz_out_str(NULL, 10, i);
          printf(", ");
          mpz_out_str(NULL, 10, j);
          printf(")=");
          ackermann(i, j, a);
          mpz_out_str(NULL, 10, a);
          printf("\n");
          mpz_add_ui(j, j, 1);
        }
      mpz_add_ui(i, i, 1);
    }
}

int main()
{
  mpz_t v;
  mpz_t dix;
  mpz_t vingt;
  mpz_init(v);
  mpz_init_set_ui(dix, 10);
  mpz_init_set_ui(vingt, 20);
  puissance(dix, vingt, v);
  afficheTableMultiplication(v, 10);
  afficheAckermann(4, 15);
  return 0;
}
```

## A.20 Algorithme d'Euclide étendu

```
#include<stdio.h>
#include<gmp.h>

/*--------------------------------------------------------------------------*/

/*
  Calcule u et v tels que au + bv = pgcd, ou pgcd est le pgcd de a et b.
  Toutes les variables doivent etre initialisees.
 */

void bezout(mpz_t a, mpz_t b, mpz_t u, mpz_t v, mpz_t pgcd)
{
  if (!mpz_cmp_ui(b, 0))
    {
      mpz_set(pgcd, a);
      mpz_set_ui(u, 1);
      mpz_set_ui(v, 0);
      printf("1 * ");
      mpz_out_str(NULL, 10, a);
      printf(" + 0 * 0 = ");
      mpz_out_str(NULL, 10, a);
    }
  else
    {
      mpz_t quotient;
      mpz_t reste;
      mpz_t uRec;
      mpz_t vRec;
      mpz_t vFoisQuotient;
      mpz_init(quotient);
      mpz_init(reste);
      mpz_init(uRec);
      mpz_init(vRec);
      mpz_init(vFoisQuotient);
      mpz_tdiv_qr(quotient, reste, a, b);
      mpz_out_str(NULL, 10, a);
      printf(" = ");
      mpz_out_str(NULL, 10, quotient);
      printf(" * ");
      mpz_out_str(NULL, 10, b);
      printf(" + ");
      mpz_out_str(NULL, 10, reste);
      printf("\n");
      bezout(b, reste, uRec, vRec, pgcd);
      mpz_out_str(NULL, 10, uRec);
      printf(" * ( ");
      mpz_out_str(NULL, 10, b);
      printf(" ) + ");
      mpz_out_str(NULL, 10, vRec);
      printf(" * ( ");
      mpz_out_str(NULL, 10, a);
      printf(" - ");
      mpz_out_str(NULL, 10, quotient);
      printf(" * ");
      mpz_out_str(NULL, 10, b);
      printf(") = ");
      mpz_out_str(NULL, 10, pgcd);
      printf("\n");
      mpz_set(u, vRec);
      mpz_mul(vFoisQuotient, vRec, quotient);
      mpz_sub(v, uRec, vFoisQuotient);
      mpz_out_str(NULL, 10, u);
      printf(" * ");
      mpz_out_str(NULL, 10, a);
      printf(" + ");
      mpz_out_str(NULL, 10, v);
      printf(" * ");
      mpz_out_str(NULL, 10, b);
      printf(" = ");
      mpz_out_str(NULL, 10, pgcd);
      mpz_clear(quotient);
      mpz_clear(reste);
      mpz_clear(uRec);
      mpz_clear(vRec);
      mpz_clear(vFoisQuotient);
    }
  printf("\n");
}
```

```
/*-------------------------------------------------------------------------*/

/*
  Implementation de la fonction phi definie dans le sujet
  phi(a, b, 0) = b et phi(a, b, n) = phi(a + b, b, n-1).
  Tous les parametres doivent etre initialises.
*/

void phi(mpz_t a, mpz_t b, mpz_t n, mpz_t res)
{
  if (!mpz_cmp_ui(n, 0))
    {
      mpz_set(res, b);
    }
  else
    {
      mpz_t aPlusB;
      mpz_t nMoinsUn;
      mpz_init(aPlusB);
      mpz_init(nMoinsUn);
      mpz_add(aPlusB, a, b);
      mpz_sub_ui(nMoinsUn, n, 1);
      phi(aPlusB, a, nMoinsUn, res);
      mpz_clear(aPlusB);
      mpz_clear(nMoinsUn);
    }
}

/*-------------------------------------------------------------------------*/

/*
  Place dans res le n-eme nombes de Fibonacci.
  Tous les parametres doivent etre initialises.
*/

void fibo(mpz_t n, mpz_t res)
{
  mpz_t zero;
  mpz_t un;
  mpz_init_set_ui(zero, 0);
  mpz_init_set_ui(un, 1);
  phi(un, zero, n, res);
  mpz_clear(zero);
  mpz_clear(un);
}

/*-------------------------------------------------------------------------*/

int main()
{
  mpz_t u;
  mpz_t v;
  mpz_t p;
  mpz_t cent;
  mpz_t centUn;
  mpz_t Fcent;
  mpz_t FcentUn;
  mpz_init(u);
  mpz_init(v);
  mpz_init(p);
  mpz_init_set_ui(cent, 100);
  mpz_init_set_ui(centUn, 101) ;
  mpz_init(Fcent);
  mpz_init(FcentUn);
  fibo(cent, Fcent);
  fibo(centUn, FcentUn);
  bezout(FcentUn, Fcent, u, v, p);
  mpz_clear(u);
  mpz_clear(v);
  mpz_clear(p);
  mpz_clear(cent);
  mpz_clear(centUn);
  mpz_clear(Fcent);
  mpz_clear(FcentUn);
  return 0;
}
```

## A.21 Espaces Quotients

```
#include<stdio.h>
#include<gmp.h>

/*-------------------------------------------------------------------------*/

/*
  Calcule u et v tels que au + bv = pgcd, ou pgcd est le pgcd de a et b.
  Toutes les variables doivent etre initialisees.
 */

void bezout(mpz_t a, mpz_t b, mpz_t u, mpz_t v, mpz_t pgcd)
{
  if (!mpz_cmp_ui(b, 0))
    {
      mpz_set(pgcd, a);
      mpz_set_ui(u, 1);
      mpz_set_ui(v, 0);
    }
  else
    {
      mpz_t quotient;
      mpz_t reste;
      mpz_t uRec;
      mpz_t vRec;
      mpz_t vFoisQuotient;
      mpz_init(quotient);
      mpz_init(reste);
      mpz_init(uRec);
      mpz_init(vRec);
      mpz_init(vFoisQuotient);
      mpz_tdiv_qr(quotient, reste, a, b);
      bezout(b, reste, uRec, vRec, pgcd);
      mpz_set(u, vRec);
      mpz_mul(vFoisQuotient, vRec, quotient);
      mpz_sub(v, uRec, vFoisQuotient);
      mpz_clear(quotient);
      mpz_clear(reste);
      mpz_clear(uRec);
      mpz_clear(vRec);
      mpz_clear(vFoisQuotient);
    }
}

/*-------------------------------------------------------------------------*/

int trouveInverse(mpz_t valeur, mpz_t module, mpz_t inverse)
{
  mpz_t u, v, pgcd;
  int res = 1;
  mpz_init(u);
  mpz_init(v);
  mpz_init(pgcd);
  bezout(valeur, module, u, v, pgcd);
  if (!mpz_cmp_ui(pgcd, 1))
      mpz_set(inverse, u);
  else
      res = 0;
  mpz_clear(u);
  mpz_clear(v);
  mpz_clear(pgcd);
  return res;
}

/*-------------------------------------------------------------------------*/

void etudieEspaceQuotient(mpz_t n)
{
  mpz_t i, phi, inverseI;
  mpz_init(i);
  mpz_init(phi);
  mpz_init(inverseI);
  mpz_set_ui(i, 0);
  mpz_set_ui(phi, 0);
  printf("Z/");
  mpz_out_str(NULL, 10, n);
  printf("Z : ");
  while(mpz_cmp(i, n))
    {
      printf("\n");
```

```
      mpz_out_str(NULL, 10, i);
      if (trouveInverse(i, n, inverseI))
        {
          printf(" a pour inverse ");
          mpz_add_ui(phi, phi, 1);
          mpz_mod(inverseI, inverseI, n);
          mpz_out_str(NULL, 10, inverseI);
        }
      else
        printf(" (non inversible) ");
      mpz_add_ui(i, i, 1);
    }
  printf("\nOn a phi(");
  mpz_out_str(NULL, 10, n);
  printf(") = ");
  mpz_out_str(NULL, 10, phi);
  printf("\n");
  mpz_add_ui(phi, phi, 1);
  if (!mpz_cmp(n, phi))
    printf("Cet ensemble est un corps\n");
  else
    printf("Cet ensemble n est pas un corps\n");
}

/*-------------------------------------------------------------------------*/

int main()
{
  mpz_t test;
  mpz_init(test);
  mpz_init_set_ui(test, 20);
  etudieEspaceQuotient(test);
  mpz_clear(test);
  return 0;
}
```

# Annexe B

# Rappels mathématiques

## B.1 Sommations

Parmi les outils mathématiques permettant d'évaluer la complexité d'un algorithme se trouvent les sommations. Commençons par rappeler quelques propriétés :

1. $\sum_{i=p}^{n} 1 = n - p + 1$
2. $\sum_{i=p}^{n} u_i = (\sum_{i=p}^{n-1} u_i) + u_n = u_p + \sum_{i=p+1}^{n} u_i$
3. $\sum_{i=p}^{n} (a.u_i) = a \sum_{i=p}^{n} u_i$
4. $\sum_{i=p}^{n} (a_i + b) = \sum_{i=p}^{n} a_i + \sum_{i=p}^{n} b = (\sum_{i=p}^{n} a_i) + (n - p + 1)b$
5. $\sum_{i=p}^{n} (a_i + b_i) = \sum_{i=p}^{n} a_i + \sum_{i=p}^{n} b_i$
6. $\sum_{i=p}^{n} u_i = \sum_{i=p+1}^{n+1} u_{i-1} = \sum_{i=p-1}^{n-1} u_{i+1}$
7. $\sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} = \sum_{j=1}^{m} \sum_{i=1}^{n} u_{ij}$

Bien que le $\sum$ soit prioritaire sur le $+$ mais pas sur le $\times$, il usuel pour éviter toute confusion de rajouter des parenthèses, même inutiles. Parmi les sommes que vous devez maîtriser parfaitement figurent les deux suivantes :

1. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (série arithmétique)
2. $\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$ (série géométrique)

On se ramène souvent à ce type de formule en algorithmique.

### B.1.1 Exercices

Calculer les sommes suivantes :

1. $\sum_{i=1}^{n} 2i + 1$

2. $\sum_{i=1}^{n}(i+1)^2$
3. $\sum_{i=1}^{n}(\sum_{j=1}^{i} j)$
4. $\sum_{i=1}^{n}(\sum_{j=i}^{n} j)$
5. $\sum_{i=p}^{n} i$
6. $\sum_{i=p}^{n} r^i$

## B.2 Ensembles

### B.2.1 Définition

**Définition B.2.1** *Un **ensemble** est un regroupement en un tout d'objets. Etant donné un ensemble $E$, les objets regroupés dans $E$ sont appelés **éléments** de $E$.*

On définit un ensemble notament en énumérant ses éléments et en les séparant par des virgules. Par exemple $E = \{1, 2, 3, 4\}$ définit un ensemble $E$ contenant les **éléments** 1, 2, 3 et 4. Si $e$ est un élément de $E$, on dit alors qu'il **appartient** à $E$, noté $e \in E$.

#### Définition à l'aide d'un prédicat

On définit aussi un ensemble par une **condition d'appartenance**. Par exemple, $E = \{e|(e \in N) \wedge (e \leq 5)\}$ définit l'ensemble des éléments $e$ vérifiant $(e \in N) \wedge (e \leq 5)$, à savoir $\{0, 1, 2, 3, 4, 5\}$. Plus formellement,

**Définition B.2.2** *Soient $P(x)$ un prédicat et $E = \{x|P(x)\}$, alors*

$$x \in E \iff P(x)$$

$E = \{x|P(x)\}$ est alors l'ensemble $E$ des éléments $x$ vérifiant $P(x)$.

#### Ensemble vide

$F$ est le prédicat constant prenant toujours la valeur "faux".

**Définition B.2.3** *On note $\emptyset$ l'**ensemble vide** défini comme suit*

$$\emptyset = \{i|F\}$$

$\emptyset$ est l'ensemble auquel n'appartient aucun élément. L'ensemble vide peut être défini par tout prédicat constant étant toujours faux. Par exemple,

$$\emptyset = \{i|(i \in N) \wedge (i = i + 1)\}$$

### B.2.2 Opérations ensemblistes

On définit sur les ensembles les opérations suivantes :

**Définition B.2.4** ***L'union** de deux ensembles $A$ et $B$, notée $\cup$, est définie par $A \cup B = \{x|(x \in A) \vee (x \in B)\}$.*

Un élément appartient à $A \cup B$ s'il appartient à $A$, **ou** s'il appartient à $B$. Par exemple,

$$\{1,3,7,8\} \cup \{2,4,6,8,10\} = \{1,2,3,4,6,7,8,10\}$$

**Définition B.2.5** ***L'intersection*** *de deux ensembles $A$ et $B$, notée $\cap$, est définie par $A \cap B = \{x | (x \in A) \wedge (x \in B)\}$.*

Un élément appartient à $A \cap B$ s'il appartient à $A$, **et** s'il appartient à $B$. Par exemple,

$$\{1,3,4,6,7,9,10\} \cap \{2,4,6,8,10,12\} = \{4,10\}$$

**Définition B.2.6** ***La différence*** *de deux ensembles $A$ et $B$, notée $\setminus$ ou $-$, est définie par $A \setminus B = \{x | (x \in A) \wedge (x \notin B)\}$.*

Un élément appartient à $A \setminus B$ (ou $A - B$) s'il appartient à $A$, **mais** pas à $B$. Par exemple,

$$\{1,2,3,4,6,7,8\} \setminus \{1,2,3\} = \{4,6,7,8\}$$

On note l'union de $n$ ensembles $E_1 \cup E_2 \cup \ldots \cup E_n$ de la façon suivante :

$$\bigcup_{i=1}^{n} E_i$$

De même

$$E_1 \cap E_2 \cap \ldots \cap E_n = \bigcap_{i=1}^{n} E_i$$

### B.2.3 Cardinal

Un ensemble est infini s'il contient un nombre infini d'éléments, sinon, on dit qu'il est fini.

**Définition B.2.7** *Le **cardinal** d'un ensemble fini $E$, noté $|E|$ est le nombre d'éléments qu'il contient.*

Par exemple, $|\emptyset| = 0$, $|\{1,2,4\}| = 3$. Seul l'ensemble vide est de cardinal 0, un ensemble de cardinal 1 est appelé un **singleton**, un ensemble de cardinal 2 est appelé une **paire**.

**Propriété B.2.1** *Le cardinal d'un ensemble a les propriétés suivantes :*
- $|A \cap B| \leq |A|$
- $|A \cap B| \leq |B|$
- $|A \cup B| \geq |A|$
- $A \setminus B \leq |A|$
- $|A \cup B| \geq |B|$
- $|A \cup B| \leq |A| + |B|$
- $|A \cup B| + |A \cap B| = |A| + |B|$
- *Si* $A \cap B = \emptyset$, $|A \cup B| = |A| + |B|$
- $|\mathcal{P}(E)| = 2^{|E|}$

### B.2.4 $n$-uplets

Un **$n$-uplet** est un regroupement ordonné de $n$ éléments non nécessairement distincts. On note $X = (x_1, \ldots, x_n)$ le $n$-uplet composé des éléments $x_1, \ldots, x_n$, on les appelle **composantes** des $E$. Par exemple, $(4,8)$ est un 2-uplet, dit aussi **couple**, et $((3,2),(6,12),(0,3))$ est un **triplet** composé des trois couples $(3,2)$, $(6,12)$ et $(0,3)$.

**Produit cartésien**

**Définition B.2.8** *Le **produit cartésien de** $n$ ensembles $E_1, \ldots, E_n$, noté $E_1 \times \ldots \times E_n$ est l'ensemble de tous les $n$-uplets $(x_1, \ldots, x_n)$ tels que $\forall i \in \{1, \ldots, n\}, x_i \in E_i$.*

Par exemple,

$$\{1,2\} \times \{a,b,c\} = \{(1,a),(2,a),(1,b),(2,b),(1,c),(2,c)\}$$

Formellement,

$$E_1 \times \ldots \times E_n = \{(x_1, \ldots, x_n) | \forall i \in \{1, \ldots, n\}, x_i \in E_i\}$$

## B.2.5 Parties

**Inclusion**

**Définition B.2.9** *$A$ est **contenu** dans $B$, noté $A \subset B$ si et seulement si $\forall e \in A, e \in B$,*

En d'autres termes si tout élément de $A$ est aussi un élément de $B$. On dit aussi que $A$ est un **sous-ensemble** de $B$, que $A$ est **inclus** dans $B$, ou encore que $A$ est une **partie** de $B$.

**Parties de $E$**

**Définition B.2.10** *L'ensemble $\mathcal{P}(E)$ des parties de $E$, est défini par*

$$\mathcal{P}(E) = \{e | e \subset E\}$$

$\mathcal{P}(E)$ est donc l'ensemble de tous les ensembles inclus dans $E$. Par exemple, $\mathcal{P}(\{1,2,3\}) = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\emptyset,\{1,2,3$ on remarque que $\mathcal{P}(E)$ est un ensemble d'ensembles. Prennez note du fait que $\emptyset$ est un sous-ensemble de tous les ensembles, y compris de lui-même, et que le seul ensemble contenu dans $\emptyset$ est $\emptyset$. A votre avis, que vaut $\mathcal{P}(\emptyset)$ ? Cette question vous amènera à distinuer $\emptyset$ de $\{\emptyset\}$, qui est l'ensemble qui contient l'ensemble vide.

**Exercice**

Calculer $\mathcal{P}(\emptyset)$, $\mathcal{P}(\{1,2\})$, $\mathcal{P}(\{\emptyset\})$.

**Autre exercice**

Soit $f$ la fonction qui a un ensemble d'ensembles $F$ et un élément $x$ associe $f(F,x) = \{e \cup \{x\} | e \in F\}$. Montrer que $\mathcal{P}(E) = f(\mathcal{P}(E - \{x\}), x) \cup \mathcal{P}(E - \{x\})$. En déduire un algorithme récursif calculant $\mathcal{P}(E)$.

## B.2.6 Définition

**Définition B.2.11** *Une **relation** $f$ entre deux ensembles $A$ et $B$ est un sous-ensemble de $A \times B$.*

Etant donné deux éléments $x \in A$ et $y \in B$, si $(x,y) \in f$, alors on dit que $f$ associe $y$ à $x$. L'ensemble des éléments qui sont associés à $x$ est $\{y | (x,y) \in f\}$.

**Définition B.2.12** *Une relation $f$ entre deux ensembles $A$ et $B$ est une **application** si $\forall a \in A, |\{b|(a,b) \in f\}| = 1$*

Plus explicitement, $f$ est une application si à tout élément de $A$ est associé exactement un élément de $B$. On dit alors que $f$ est une application **de $A$ dans $B$**, ce qui se note $f : A \longrightarrow B$. On dit par abus de langage que $A$ est l'ensemble de départ et $B$ l'ensemble d'arrivée. Pour tout $a \in A$, on note $f(a)$ l'élément de $B$ qui lui est associé. On dit que $f(a)$ est **l'image** de $a$ par $f$ et $a$ **un antécédent** de $f(a)$ par $f$.

Etant donné un ensemble $E$, il existe une application, appelée **identité**, et notée $id$, telle que tout élément à pour image lui-même. Autrement dit : $id : E \longrightarrow E, x \mapsto x$. La succession de symboles $a \mapsto b$ signifie que $b$ est l'image de $a$, donc $x \mapsto x$ signifie que $x$ a pour image lui-même.

### B.2.7 Composition

Etant donnés $f : A \longrightarrow B$ et $g : B \longrightarrow C$, on définit l'application composée de $g$ et $f$, notée $g \circ f$, dite "$g$ rond $f$", et définie par $g \circ f : A \longrightarrow C, x \mapsto g(f(x))$. Autrement dit, $g \circ f$ est une application de $A$ dans $C$, qui à tout élément $x$ de $A$, associe l'élément $g(f(x))$ de $C$.

### B.2.8 Classification des applications

Soit $f : A \longrightarrow B$,

**Définition B.2.13** *$f$ est injective si*

$$\forall a \in A, \forall a' \in A, f(a) = f(a') \Rightarrow a = a'$$

$f$ est injective si deux éléments distincts ne peuvent pas avoir la même image.

**Définition B.2.14** *$f$ est surjective si*

$$\forall b \in B, \exists a \in A, f(a) = b$$

$f$ est surjective si tout élément de l'ensemble d'arrivée a un antécedent.

**Définition B.2.15** *$f$ est bijective si elle est à la fois injective et surjective.*

$f$ est bijective si tous les éléments de $A$ et de $B$ sont reliés deux à deux.

**Propriété B.2.2** *Soient $A$ et $B$ deux ensembles et $f : A \longrightarrow B$,*
- *Si $f$ est injective, alors $|A| \leq |B|$*
- *Si $f$ est surjective, alors $|A| \geq |B|$*
- *Si $f$ est bijective, alors $|A| = |B|$*

Cette propriété est très importante ! En dénombrement, lorsque le cardinal d'un ensemble est difficile à déterminer, on passe par une bijection vers un autre ensemble dont il est davantage aisé de calculer le cardinal.

### B.2.9 Application réciproque

Soit $f : A \longrightarrow B$ une application bijective.

**Théorème B.2.1** *Il existe une unique fonction $f^{-1}$, appelée **application réciproque de f**, telle que pour tous $x \in A$, $y \in B$ tels que $f(x) = y$, on ait $f^{-1}(y) = x$.*

On remarque que $f \circ f^{-1} = f^{-1} \circ f = id$.

## B.3 Raisonnements par récurrence

### B.3.1 Principe

Ce type de raisonnement ne s'applique que dans des propriétés faisant intervenir des nombres entiers. Par exemple, "Quel que soit $n \geq 0$, $n^2 - n$ est pair". Pour prouver qu'une propriété est vérifiée quelle que soit la valeur de $n$, on effectue une preuve par récurrence en procédant en deux temps :
- **l'initialisation**, On prouve que pour $n = 0$ (ou 1, ça dépend des cas), la propriété est vérifiée.
- **l'hérédité**, on prouve que si la propriété est vérifiée au rang $n$, alors elle l'est au rang $n + 1$.

### B.3.2 Exemple

Montrons par récurrence sur $n$ que "Quel que soit $n \geq 0$, $n^2 - n$ est pair" :

- **initialisation :** , $0^2 - 0$ est pair, c'est évident.
- **hérédité :** , supposons que $n^2 - n$ est pair, vérifions si $(n+1)^2 - (n+1)$ est pair lui aussi. Calculons

$$(n+1)^2 - (n+1) = n^2 + 2n + 1 - n - 1 = (n^2 - n) + 2n$$

  Comme $(n^2 - n)$ et $2n$ sont tous deux pairs, et que la somme de deux nombres pairs est paire, alors $(n+1)^2 - (n+1)$ est pair.

Demandons-nous si $2^2 - 2$ est pair, on sait d'après l'initialisation que $0^2 - 0$ est pair. Posons $n = 0$, comme la propriété est vérifiée au rang $n$, elle est, d'après l'hérédité, nécessairement vérifiée au rang $n + 1$, donc $1^2 - 1$ est pair. Posons $n = 1$, comme $1^2 - 1$ est pair, la propriété est vérifiée au rang $n$, elle est, d'après l'hérédité, nécessairement vérifiée au rang $n + 1$, donc $2^2 - 2$ est pair. On peut généraliser ce raisonnement à n'importe quelle valeur de $n$.

### B.3.3 Exercices

#### Exercice 10

Prouver par récurrence les propriétés suivantes :

1. Quel que soit $n \geq 0$, $n^2 + n + 1$ est impaire.
2. Soit $(u)$ une suite définie par $u_0 = -2$, $u_{n+1} = (1/2)u_n + 3$, alors $u_n < 6$
3. Soit $(u)$ une suite définie par $u_0 = -2$, $u_{n+1} = (1/2)u_n + 3$, alors $u_n = 6 - \frac{8}{(2^n)}$
4. Soit $(u)$ une suite définie par $u_0 = 2$, $u_{n+1} = 2u_n - n$, alors $u_n = 2^n + n + 1$
5. Pour tout $n > 0$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
6. Pour tout $n \geq 0$, $q \neq 1$, $\sum_{i=0}^{n} q^i = \frac{1 - q^{n+1}}{1 - q}$
7. Pour tout $n > 0$, $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

#### Dérivée de fonctions puissance $n$

On rappelle que $(fg)' = f'g + fg'$ et que $f^n$ est définie par $(f^n)(x) = [f(x)]^n$. Prouvez par récurrence que $\forall n \in \mathbb{N}^*$

$$(f^n)' = nf'(f^{n-1})$$

#### Formule de Leibniz

On note $f^{(n)}$ la dérivée $n$-ième de $f$. Prouvez par récurrence la formule de Leibniz : $\forall n \in \mathbb{N}^*$

$$(fg)^n = \sum_{i=0}^{n} \mathcal{C}_n^i f^{(i)} g^{(n-i)}$$

#### Dérivée de fonctions composées

On rappelle que $f \circ g$ est défini par $(f \circ g)(x) = f(g(x))$ et que $(f \circ g)' = g'(f' \circ g)$. Etant donné un ensemble $\{f_1, \ldots, f_n\}$ de $n$ fonctions, on note

$$\bigcirc_{i=1}^{n}[f_i] = f_1 \circ \ldots \circ f_n$$

la composition de ces fonctions. Démontrez par récurrence que

$$(\bigcirc_{i=1}^{n}[f_i])' = \prod_{i=1}^{n}[f_i' \circ (\bigcirc_{j=i+1}^{n}[f_j])]$$

## B.4 Analyse combinatoire

### B.4.1 Factorielles

Pour tout $n \geq 0$, le nombre $n!$, appelé "factorielle n", est défini par récurrence à l'aide des relations suivantes :
- $0! = 1$
- $n! = n \times (n-1)!$

On a donc $n! = \prod_{i=1}^{n} i$ avec le cas particulier $0! = 1$.

### B.4.2 Arrangements

**Définition B.4.1** *Soient $n$ et $p$ tels que $0 \leq p \leq n$, on note $\mathcal{A}_n^p$, et on lit "A n p" le nombre*

$$\frac{n!}{(n-p)!}$$

### B.4.3 Combinaisons

**Définition B.4.2** *Soient $n$ et $p$ tels que $0 \leq p \leq n$, on note $\mathcal{C}_n^p$, et on lit "C n p" le nombre*

$$\frac{n!}{p!(n-p)!}$$

**Propriété B.4.1** *Les égalités suivantes sont vérifiées pour tous $k$ et $n$ tels que $0 \leq k \leq n$,*
- $\mathcal{C}_n^i = \frac{1}{p!}\mathcal{A}_n^i$
- $\mathcal{C}_n^0 = 1$
- $\mathcal{C}_n^1 = n$
- $\mathcal{C}_n^p = \mathcal{C}_n^{n-p}$
- $p\mathcal{C}_n^p = n\mathcal{C}_{n-1}^{p-1}$
- $\mathcal{C}_n^p = \mathcal{C}_{n-1}^{p-1} + \mathcal{C}_{n-1}^p$
- $\sum_{i=0}^{n} \mathcal{C}_n^i = 2^n$
- $(a+b)^n = \sum_{i=0}^{n} \mathcal{C}_n^i a^i b^{n-i}$
- $\sum_{i=p-1}^{n-1} \mathcal{C}_i^{p-1} = \mathcal{C}_n^p$

### B.4.4 Triangle de Pascal

Le triangle de Pascal est un tableau triangulaire de taille infinie dont les lignes sont indicées par $n$ et les colonnes par $p$, les indices commençant à 0. Les seules case du trianles renseignées sont celles dont les indices vérifient $0 \leq p \leq n$. Les valeurs se trouvant à la ligne d'indice $n$ et la colonne d'indice $p$ du triangle de Pascal est $\mathcal{C}_n^p$, on a ainsi

| $(n,p)$ | | $p=$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| $n=$ | 0 | | 1 | | | | | | | |
| | 1 | | 1 | 1 | | | | | | |
| | 2 | | 1 | 2 | 1 | | | | | |
| | 3 | | 1 | 3 | 3 | 1 | | | | |
| | 4 | | 1 | 4 | 6 | 4 | 1 | | | |
| | 5 | | 1 | 5 | 10 | 10 | 5 | 1 | | |
| | 6 | | 1 | 6 | 15 | 20 | 15 | 6 | 1 | |
| | ... | | ... | ... | ... | ... | ... | ... | ... | ... |

Vous remarquez que la propriété $\mathcal{C}_n^0 = 1$ traduit le fait que la première colonne ne comporte que des 1. Comme chaque ligne est symétrique alors $\mathcal{C}_n^p = \mathcal{C}_n^{n-p}$. Pour chaque ligne, le premier élément est 1, donc la symétrie fait que le dernier est 1 aussi. La propriété $\mathcal{C}_n^p = \mathcal{C}_{n-1}^{p-1} + \mathcal{C}_{n-1}^p$ traduit le fait que chaque élément $\mathcal{C}_n^p$ ne se trouvant pas dans la première colonne ou sur la diagonale est la somme de celui qui est au dessus $\mathcal{C}_{n-1}^p$ et de celui qui est juste à gauche de ce dernier $\mathcal{C}_{n-1}^{p-1}$. Si on somme les éléments sur chaque ligne, on obtient des puissances successives de 2 :

| $(n,p)$ | | $p=$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | $\sum_{p=0}^{n} \mathcal{C}_n^p$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n=$ | 0 | | 1 | | | | | | | | $2^0$ |
| | 1 | | 1 | 1 | | | | | | | $2^1$ |
| | 2 | | 1 | 2 | 1 | | | | | | $2^2$ |
| | 3 | | 1 | 3 | 3 | 1 | | | | | $2^3$ |
| | 4 | | 1 | 4 | 6 | 4 | 1 | | | | $2^4$ |
| | 5 | | 1 | 5 | 10 | 10 | 5 | 1 | | | $2^5$ |
| | 6 | | 1 | 6 | 15 | 20 | 15 | 6 | 1 | | $2^6$ |
| | ... | | ... | ... | ... | ... | ... | ... | ... | ... | ... |

Cette propriété se traduit $\sum_{i=0}^{n} \mathcal{C}_n^i = 2^n$. L'identité remarquable $(a+b)^n$ s'écrit à l'aide des coefficients su triangle de Pascal. Par exemple,

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

Vous remarquez que le développement de $(a+b)^3$ s'ecrit avec un polynome à deux variable dont la somme des exposants de chaque terme est 3, chaque terme est donc de la forme $a^i b^{3-i}$. Les coefficients devant chaque termes sont issus de la ligne d'indice 3 du triangle de Pascal, à savoir $(1,3,3,1)$, autrement dit $(\mathcal{C}_3^0, \mathcal{C}_3^1, \mathcal{C}_3^2, \mathcal{C}_3^3)$. Donc

$$(a+b)^3 = \sum_{i=0}^{3} \mathcal{C}_3^i a^i b^{n-i}$$

On a plus généralement que

$$(a+b)^n = \sum_{i=0}^{n} \mathcal{C}_n^i a^i b^{n-i}$$

Par exemple,

$$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$$

Le soin de le démontrer par récurrence vous est laissé.

### B.4.5 Liens avec les ensembles

Le nombre de façons d'ordonner $n$ éléments distincts est donnée par $n!$ (la preuve vous est laissé en exercice).

**Propriété B.4.2** *Le nombre de sous-ensembles à $k$ éléments d'un ensemble $E$ à $n$ éléments, à savoir $|\{x|(x \subset E) \wedge (|x| = k)\}|$ est donné par $\mathcal{C}_n^k$.*

On le démontre par récurrence sur $n$,

- si $n = 0$, le seul ensemble à 0 éléments est $\emptyset$, son seul sous-ensemble est $\emptyset$, donc

$$|\{x|(x \subset \emptyset) \wedge (|x| = 0)\}| = |\{\emptyset\}| = 1 = \mathcal{C}_0^0$$

- Supposons que pour tout ensemble $E'$ à $n-1 \geq 0$ éléments, et tout $k' \in \{0, \ldots, n-1\}$,

$$|\{x|(x \subset E') \wedge (|x| = k')\}| = \mathcal{C}_{n-1}^{k'}$$

  Considérons un ensemble $E$ à $n \geq 1$ éléments et une valeur $k \in \{0, \ldots, n\}$, montrons que le cardinal de

$$P = \{x|(x \subset E) \wedge (|x| = k)\}$$

  est $\mathcal{C}_n^k$

– Si $k \in \{1, \ldots, n-1\}$, choisissons un élément arbitraire $e \in E$, cet élément existe nécessairement car $|E| = n \geq 1$. On décompose $\{x|(x \subset E) \wedge (|x| = k)\}$ en deux ensembles $P_1$ et $P_2$. $P_1$ est l'ensemble des sous-ensembles de $E$ à $k$ éléments qui contiennent $e$, défini par

$$P_1 = \{x|(x \subset E) \wedge (|x| = k) \wedge (e \in x)\}$$

$P'$ est celui des sous-ensembles à $k$ éléments qui ne contiennent pas $e$,

$$P_2 = \{x|(x \subset E) \wedge (|x| = k) \wedge (e \notin x)\}$$

On remarque que $P_1 \cup P_2 = P$ et que $P_1 \cap P_2 = \emptyset$, donc $|P| = |P_1| + |P_2|$. Comme $P_2$ est l'ensemble des sous-ensembles à $k$ éléments de $E - \{e\}$, on le redéfinit de la sorte :

$$P_2 = \{x|(x \subset E - \{e\}) \wedge (|x| = k)\}$$

Comme $|E - \{e\}| = n - 1$ et $k \in \{1, \ldots, n-1\}$, alors par hypothèse de récurrence, $|P_2| = \mathcal{C}_{n-1}^k$. Par ailleurs, chaque élement $x$ de $P_1$ se décompose de la sorte : $x = x' \cup \{e\}$ où $x'$ est un ensemble à $(k-1)$ éléments ne contenant pas $e$. Donc

$$P_1 = \{x' \cup \{e\}|(x' \subset E - \{e\}) \wedge (|x'| = k - 1)\}$$

Il a donc autant d'élément dans $P_1$ que de sous-ensembles à $k-1$ éléments de $E-\{e\}$. Comme $|E-\{e\}| = n-1$ et $(k-1) \in \{0, \ldots, n-2\}$, alors par hypothèse de récurrence, on a $|P_1| = \mathcal{C}_{n-1}^{k-1}$. De ce fait $|P| = \mathcal{C}_{n-1}^k + \mathcal{C}_{n-1}^{k-1} = \mathcal{C}_n^k$.

– Dans le cas où $k = 0$ et $n \geq 1$, on remarque que

$$P = \{x|(x \subset E) \wedge (|x| = 0)\} = \{\emptyset\}$$

et que par conséquent $|P| = |\{\emptyset\}| = 1 = \mathcal{C}_n^0$

– Il nous reste à traiter le cas où $k = n$ et $n \geq 1$, on remarque que

$$P = \{x|(x \subset E) \wedge (|x| = |E|)\} = \{E\}$$

et que par conséquent $|P| = |\{E\}| = 1 = \mathcal{C}_n^n$.

On a bien

$$\{x|(x \subset E) \wedge (|x| = k)\} = \mathcal{C}_n^k$$

Pour tout $k \in \{0, \ldots, n\}$.

Etant donné un ensemble $E$ à $n$ éléments, dénombrons les parties de $E$, ce qui revient à calculer $|\mathcal{P}(E)|$. Pour ce faire, on décompose $\mathcal{P}(E)$ en $n+1$ sous-ensembles $\mathcal{P}_k(E)$, $k \in \{0, \ldots, n\}$, définis comme suit

$$\mathcal{P}_k(E) = \{e|(e \subset E) \wedge (|e| = k)\}$$

On remarque que pour tout $e \subset E$, $e \in \mathcal{P}_{|e|}(E)$, donc $\mathcal{P}(E) \subset \bigcup_{k=0}^{n} \mathcal{P}_k(E)$. Réciproquement, pour tout $k \in \{0, \ldots, n\}$, tout élément de $\mathcal{P}_k(E)$ est, par définition, une partie de $E$, donc $\bigcup_{k=0}^{n} \mathcal{P}_k(E) \subset \mathcal{P}(E)$. On déduit de cette double inclusion que

$$\mathcal{P}(E) = \bigcup_{k=0}^{n} \mathcal{P}_k(E)$$

Deux ensembles $\mathcal{P}_k(E)$ et $\mathcal{P}_{k'}(E)$ tels que $k \neq k'$ sont nécessairements disjoints(si ça vous semble pas évident, cherchez un contre-exemple et vous comprendrez pourquoi...). Donc

$$|\bigcup_{k=0}^{n} \mathcal{P}_k(E)| = \sum_{k=0}^{n} |\mathcal{P}_k(E)|$$

Si cette propriété ne vous parle pas, démontrez-la par récurrence en utilisant le fait que si $A \cap B = \emptyset$, alors $|A \cap B| = |A| + |B|$. Nous avons démontré que $\forall k \in \{0, \ldots, n\}, |\mathcal{P}_k(E)| = \mathcal{C}_n^k$. Donc $\sum_{k=0}^{n} |\mathcal{P}_k(E)| = \sum_{k=0}^{n} \mathcal{C}_n^k = 2^n$. Le nombre de parties d'un ensemble à $n$ éléments est donc $2^n$. Il vous est possible, de vérifier ce résultat par récurrence, le raisonnement est analogue à celui employé pour prouver que $|\{x|(x \subset E) \wedge (|x| = k)\}| = \mathcal{C}_n^k$, mais en plus facile.

## B.5 Exercices récapitulatifs

### B.5.1 Echauffement

Démontrer avec la méthode la plus appropriée les propositions suivantes :

- $C_n^0 = 1$
- $C_n^1 = n$
- $C_n^p = C_n^{n-p}$
- $pC_n^p = nC_{n-1}^{p-1}$
- $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$
- $\sum_{i=0}^{n} C_n^i = 2^n$
- $(a+b)^n = \sum_{i=0}^{n} C_n^i a^i b^{n-i}$
- $\sum_{i=p-1}^{n-1} C_i^{p-1} = C_n^p$

### B.5.2 Formule de Poincaré

Le but de cet exercice est de prouver la formule de Poincaré :

$$|\bigcup_{i=1}^{n} E_i| = \sum_{i=1}^{n} (-1)^{i+1} \sum_{e \in I_i^n} (\bigcap_{i \in e} |E_i|)$$

En notant $I_k^n$ l'ensemble des sous-ensembles de $\{1, \ldots, n\}$ à $k$ éléments, formellement : $I_k^n = \{e \subset \{1, \ldots, n\}\ tel\ que\ |e| = k\}$

1. Réecrivez la formule de Poincaré avec $n = 1$
2. Réecrivez la formule de Poincaré avec $n = 2$
3. Soient $A = \{1, 3, 5, 6\}$ et $B = \{1, 2, 3, 4\}$. Calculez $|A \cup B|$ en utilisant la propriété $|A \cup B| = |A| + |B| - |A \cap B|$.
4. Réecrivez la formule de Poincaré avec $n = 3$
5. Soit $C = \{6, 7, 8\}$, utilisez la formule de Poincaré pour calculer $|A \cup B \cup C|$.
6. On considère $n+1$ ensembles $\{E_1, \ldots E_{n+1}\}$. Supposons la formule de Poincaré vérifiée vérifiée au rang $n$. Donnez alors une relation entre

   $$|\bigcup_{i=1}^{n+1} E_i|$$

   et

   $$\sum_{i=1}^{n} (-1)^{i+1} \sum_{e \in I_i^n} (\bigcap_{i \in e} |E_i|)$$

7. Utilisez les résultats des questions précédentes pour prouver par récurrence la formule de Poincaré.

### B.5.3 La moulinette à méninges

Soient $A$ et $B$ deux ensembles finis de cardinaux respectifs $m$ et $n$.

1. Combien existe-t-il de relations entre $A$ et $B$ ? (Bijection avec les parties de l'ensemble $A \times B$)
2. Combien existe-t-il d'applications de $A$ dans $B$ ? (Raisonnez par récurrence)
3. Combien existe-t-il d'applications bijectives de $A$ dans $B$ ? (une bijection)
4. Combien existe-t-il de $k$-uplets d'éléments de $A$ ? (démerdez-vous)
5. Combien existe-t-il de $k$-uplets d'éléments distincts de $A$ ?
6. Combien existe-t-il d'applications injectives de $A$ dans $B$ ? (raisonnez par récurrence)
7. Combien existe-t-il de façon de partitionner un ensemble $E$ de cardinal $n$ en 3 sous-ensembles ? C'est-à-dire de déterminer 3 ensembles $A$, $B$ et $C$ tels que $A \cup B \cup C = E$, $A \cap B = \emptyset$, $A \cap C = \emptyset$ et $B \cap C = \emptyset$. (bijection)
8. Une application est croissante au sens large si $\forall x, y \in A, x < y \Longrightarrow f(x) \leq f(y)$. Combien existe-t-il d'applications croissantes au sens large de $A$ dans $B$ (supposés ordonnés) ? (une bijection)
9. Une application est croissante au sens strict si $\forall x, y \in A, x < y \Longrightarrow f(x) < f(y)$. Combien existe-t-il d'applications croissantes au sens large de $A$ dans $B$ ? (plus difficile, bijection...)
10. Combien existe-t-il de façon de partitionner un ensemble $E$ de cardinal $n > 3$ en 3 sous-ensembles non-vides ? (bijections + poincaré)
11. Soit $f : A \longrightarrow A$, un point fixe de $f$ est un élément $x$ de $A$ tel que $f(x) = x$. Combien existe-t-il de bijection sans point fixe ? (vraiment difficile, utilisez la formule de Poincaré puis le binôme de Newton)

# Bibliographie


[1] Simon Singh. *The code book : the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography*. 1999.

[2] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to algorithms*. Dunod, 1992.

[3] Chrétienne P. Beauquier D., Berstel J. *Eléments d'algorithmique*. Masson, 1992.

[4] `http://webpages.ull.es/users/jriera/Docencia/AVL/AVLtreeapplet.htm`.

[5] Vasek Chvatal. *Linear Programming*. Freeman, 1983.

[6] `http://www.gnu.org/software/glpk/`.

[7] `http://www.lmbe.seu.edu.cn/CRAN/doc/packages/glpk.pdf`.

[8] Johnson D Garey M. *Computers and Intractability*. 1979.

[9] *Hackademy Journal*.

[10] *Hackademy Journal*.

[11] *Hackademy Journal*.

[12] `http://www.gnupg.org/(fr)/documentation/guides.html`.

[13] `http://www.gnupg.org/gph/en/manual.pdf`.

[14] `http://gmplib.org/`.

[15] `http://gmplib.org/#DOC`.