Algorithmique et application Java

Denis Cornaz [1]

1er avril 2014

1. LAMSADE, Université Paris-Dauphine, Place du Maréchal de Lattre de Tassigny 75775 Paris Cedex 16, FRANCE. (e-mail : denis.cornaz@dauphine.fr)

Table des matières

Résumé

Ce cours, Informatique 4 du MIDO/DUMI2E 2ième année de l'université Paris-Dauphine, s'adresse à des étudiants initiés à l'algorithmique et à la programmation Java, c'est-à-dire ayant déjà programmé quelques algorithmes en Java d'au-moins 20 lignes, et ayant un bagage mathématique classique.

Le but pour l'étudiant est l'acquisition des techniques fondamentales de l'analyse algorithmique et de la conception d'algorithmes, motivées par le comportement en pratique de l'exécution de programmes en Java. En particulier, nous étudierons les techniques permettant un gain de complexité ; celles liées par exemple à l'utilisation de structure de données arborescentes ou à la récursivité.

La pertinence des concepts introduits sera confrontée à la réalité via des allers et retours entre la théorie et l'exécution sur machine.

Chapitre 1

Complexité des algorithmes

Toute l'algorithmique repose sur l'idée qu'il y a des algorithmes meilleurs que d'autres, indépendamment des aspects technologiques, et que l'on peut ainsi les classer théoriquement selon divers critères de performance. Le concept central est la *complexité*, détaillé dans la prochaine section, c'est-à-dire la prévision du temps d'exécution d'un algorithme en fonction de la taille des paramètres en entrée.

1.1 Introduction et notation asymptotique

Pour toute fonction $g(n) : \mathbb{N} \to \mathbb{N}$, on note :

$$O(g(n)) = \{f(n) : 0 \leq f(n) \leq cg(n), \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

Le point de départ du concept de complexité d'un algorithme est :

Point 1 *Si un algorithme n'a aucun paramètre en entrée, alors sa complexité est* $O(1)$.

Ce postulat implique par-exemple que, soit une classe possédant trois méthodes `A()`, `B()` et `C()` suivantes, elles ont la même complexité :

```
public static void A (){
  int a=0,b=1;
  for(int i=0; i<10; ++i)
     a += b;
}

public static void B (){
  int a=0,b=100;
  for(int i=0; i<100; ++i)
     a += b;
}

public static void C (){
  for(int i=0; i<10000; ++i){
     A();
     A();
     B();
  }
}
```

Le nombre d'additions effectuées par un appel à `A()` est $O(1)$, de même que le nombre d'additions effectuées par un appel à `B()` ou à `C()`.

Point 2 *Si un algorithme* `A` *consiste en deux appels à un algorithme* `B`*, alors les deux algorithmes ont la même complexité. Ce qui signifie que la complexité ne se mesure qu'à une constante multiplicative près.*

Dans les méthodes surchargées suivantes, `A(int n)` fera n boucles constantes, chacune de complexité $O(1)$, `B(int n, int m)` fera nm boucles constantes et `C(int n, int m)` fera $2nm$ boucles constantes. Mais les méthodes `B(int n, int m)` et `C(int n, int m)` ont la même complexité $O(mn)$.

```
public static void A (int n){
  for(int i=0; i<n; ++i)
    A();
}

public static void B (int n, int m){
  for(int i=0; i<m; ++i){
    A(n);
  }
}

public static void C (int n, int m){
  for(int i=0; i<m; ++i){
    A(n);
    A(n);
  }
}
```

Pour toute fonction $g(n)$, on note :

$$\Omega(g(n)) = \{f(n) : 0 \leq cg(n) \leq f(n), \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

On note aussi :

$$\Theta(g(n)) = \{f(n) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

Clairement, le nombre d'exécutions de la méthode `A()` par un appel de `D1(int n)` est $n^2 = \Theta(n^2)$, Pour un appel de `D2(int n)`, `A()` sera exécutée $\sum_{i=1}^{i=n} i = \frac{n(n+1)}{2}$ fois, soit $\Theta(n^2)$ fois, en prenant $c_1 = 1/2$, $c_2 = 1$ et $n_0 = 0$.

```
public static void D1(int n){
 for(int i=0;i<n;++i)
    for(int j=0;j<n;++j)
      A();
}
public static void D2(int n){
 for(int i=0;i<n;++i)
    for(int j=i;j<n;++j)
      A();
}
```

En théorie, les méthodes `D1(int n)` et `D2(int n)` sont classées dans la même catégorie, celle des algorithmes $\Theta(n^2)$.

Les notations asymptotiques O, Ω, Θ s'étendent aux fonctions de plusieurs variables, par exemple :

$$O(g(n,m)) = \{f(n,m) : 0 \leq f(n,m) \leq cg(n,m), \exists c > 0, \exists n_0, m_0 \in \mathbb{N}, \forall n \geq n_0, \forall n \geq m_0\}.$$

Définition 1 *La* complexité d'un algorithme en fonction du vecteur de paramètres n dépendants de l'entrée *est*

- Au-plus $O(g(n))$ *si son exécution se décompose en au-plus* $O(g(n))$ *exécutions d'algorithmes constants ;*
- Au-moins $\Omega(g(n))$ *si on a une décomposition en au-moins* $\Omega(g(n))$ *algorithmes constants ;*
- Exactement $\Theta(g(n))$ *si son éxécution se décompose en au-plus* $O(g(n))$ *et au-moins* $\Omega(g(n))$ *exécutions d'algorithmes constants.*

Avec $g : \mathbb{N}^k \to \mathbb{R}_+$ *où* k *est le nombre de paramètres entiers choisis pour caractériser l'entrée.*

Clairement, la complexité de `C(int n)` ci-dessous est au-plus $O(n^2)$. Montrons qu'elle est en fait $\Theta(n)$.

```
public static void D(boolean[] Tab){
  int j=0;
  while(Tab[j]==true){
    Tab[j]=false;
    j++;
  }
  Tab[j]=true; //j<Tab.length
}
public static void C(int n){
  boolean[] Tab;
  Tab = new boolean[n];
  for(int i=0;i<n;++i)
    D(Tab);
}
```

Clairement la complexité de `C(int n)` est au-moins $\Omega(n)$, il suffit donc de montrer qu'elle est au-plus $O(n)$. Elle est $O(t(n))$ où $t(n)$ est le nombre total de basculements de `Tab[j]` de `true` à `false` dans la boucle **while**, ou de `false` à `true`, à chacun des n appels à `D(boolean[] Tab)`. Ainsi, $t(n) = \sum_{j=0}^{j=n-1} b_n(j)$, où $b_n(j)$ est le nombre de basculements de `Tab[j]`. Soit $T_{ij} \in \{0, 1\}$ tel que $T_{ij} = 1$ si et seulement si `Tab[j]=true` après le $i$ème appel de `D(boolean[] Tab)` (Java initialise `Tab[j]=false` pour $j = 0, \ldots, n-1$) :

$i \setminus j$	0	1	2	3	...	n
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	1	1	0	0	0	0
4	0	0	1	0	0	0
5	1	0	1	0	0	0

Soit k le plus petit indice j tel que $T_{ij} = 0$:

$i \setminus j$	0	1	2	...	k	...
i	1	1	1	1	0	x
$i+1$	0	0	0	0	1	x

On a alors

$$T_{i+1j} - T_{ij} = \begin{cases} -1 & \text{pour } j < k, \\ 1 & \text{pour } j = k, \text{ et} \\ 0 & \text{pour } j > k. \end{cases}$$

On peut montrer par induction sur i, que $i = \sum_{j=0}^{j=n-1} T_{ij} 2^j := f(i)$. Clairement $f(0) = 0$. Il suffit donc montrer que $f(i+1) - f(i) = 1$. Donc $f(i+1) - f(i) = \sum_{j=0}^{j=n-1} 2^j (T_{i+1j} - T_{ij}) = 2^k - \sum_{j=0}^{j=k-1} 2^j = 1$. (Par induction sur n on a $2^n - 1 = \sum_{j<n} 2^j$, en effet, $2^{n+1} - 1 = 2(2^n - 1) + 1 = 1 + \sum_{0<j<n+1} 2^j = \sum_{j<n+1} 2^j$.)

On peut montrer, par induction sur n, que $b_n(j) = \lfloor \frac{n}{2^j} \rfloor$ qui est clairement vraie pour $n = 0\ \forall j$ et $\forall n\ j = 0$. Remarquons que, ci-dessous, l'égalité de gauche est vraie, il suffit donc de montrer que celle de droite est vraie.

$$b_{i+1}(j) = \begin{cases} 1 + b_i(j) & \text{si } j \leq k \text{ et} \\ b_i(j) & \text{si } j > k, \end{cases} \quad \text{et } \left\lfloor \frac{i+1}{2^j} \right\rfloor = \begin{cases} 1 + \left\lfloor \frac{i}{2^j} \right\rfloor & \text{si } j \leq k \text{ et} \\ \left\lfloor \frac{i}{2^j} \right\rfloor & \text{si } j > k. \end{cases}$$

Donc on a $j > 0$, $i = f(i) = 2^k - 1 + q2^k$ et $i + 1 = (q + 1)2^k$, $\exists q$. Donc si $j \leq k$, on a $k > 0$ et $i = (2q + 1)2^{k-1} + 2^{k-1} - 1 = \ldots$, et il s'ensuit que, $\exists q'$, $i = q'2^j + 2^j - 1$ pour $j \leq k$. Donc la première égalité des deux de droite est vraie. Pour $j > k$, on a $i = f(i) = 2^k - 1 + \sum_{l=k+1}^{n-1} T_{il}2^l$, donc 2^j ne divise ni i ni $i + 1$, d'où la deuxième égalité.

La preuve finale découle donc du calcul suivant :

$$\begin{aligned} t(n) &= \sum_{j=0}^{j=n-1} \left\lfloor \frac{n}{2^j} \right\rfloor \\ &\leq n \sum_{j=0}^{\infty} \frac{1}{2^j} \\ &= 2n. \end{aligned}$$

(Car par induction sur n on a $1 - \frac{1}{2^n} = \sum_{j=1}^{n} \frac{1}{2^j}$, en effet, $1 - \frac{1}{2^{n+1}} = \frac{1}{2} + \frac{1}{2}(1 - \frac{1}{2^n}) = \frac{1}{2} + \frac{1}{2}(\sum_{j=1}^{n} \frac{1}{2^j}) = \sum_{j=1}^{n+1} \frac{1}{2^j}$.)

1.2 Complexités au pire cas et en moyenne

Le temps d'exécution d'un algorithme dépend de la nature des données en entrée.

Définition 2 *Etant donné un algorithme, soit $\mathcal{A}$ l'ensemble de toutes les entrées possibles pour cet algorithme. Si la complexité de l'algorithme, avec une entrée $A \in \mathcal{A}$ de taille n, est $\Theta(g(n, A))$, alors sa* complexité au pire cas *est $\Theta(g(n))$ avec*

$$g(n) := \max_{A \in \mathcal{A}} g(n, A),$$

et sa complexité moyenne *est $\Theta(g(n))$ avec*

$$g(n) := \sum_{A \in \mathcal{A}} \frac{g(n, A)}{|\mathcal{A}|}.$$

Par défaut, on exprime la complexité au pire cas des algorithmes. On peut remplacer "$\Theta(\cdot)$" par "au-plus $O(\cdot)$" ou par "au-moins $\Omega(\cdot)$".

La procédure `D(boolean[] Tab)` que nous avons vu a une complexité $\Theta(n)$ au pire cas et $O(1)$ en moyenne (avec n égale `Tab.length`).

Typiquement le nombre d'opérations considérées constantes (comme la comparaison d'entiers) de **Algorithm 1** est $\Theta(n + t(A))$ où $t(A)$ est le nombre de passages dans la boucle **while**, qui dépend du tableau A à trier et pas uniquement de sa taille n.

La validité de **Algorithm 1**, c'est-à-dire le fait qu'après son exécution, A soit trié (dans l'ordre croissant), découle de l'invariant : *soit A_j le tableau A avant l'entrée de la boucle* **for** *(ainsi A_2 est le tableau A d'origine), alors d'une part A_j est une permutation de A, d'autre part, le sous-tableau $A_j[1 \ldots j-1]$ est trié.* Trivialement c'est vrai pour $j = 2$. Supposons le vrai jusqu'à j. Soit i la valeur de cet indice à la sortie de la boucle **while**, on a $A_{j+1}[1 \ldots i] = A_j[1 \ldots i]$ et $A_{j+1}[i+2 \ldots j] = A_j[i+1 \ldots j-1]$. De par la dernière instruction on a $A_{j+1}[i] \leq A_{j+1}[i+1] = A_j[j] \leq A_{j+1}[i+2]$. Donc l'invariant est vrai et la validité démontrée.

Algorithm 1 TRI-INSERTION(A)

for $j \leftarrow 2$ à n **do**
  $clé \leftarrow A[j]$
  $i \leftarrow j-1$
  while $i > 0$ et $A[i] > clé$ **do**
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i-1$
  $A[i+1] \leftarrow clé$

Si $A[i] > A[j]$ pour tout $1 \leq i < j \leq n$, alors $t(A) = \sum_{j=2}^{j=n} j - 1 = \Theta(n^2)$, c'est ce qui se produit si le tableau en entrée A est trié dans l'ordre décroissant et que les éléments de A sont tous différents. Si $A[i] \leq A[j]$ pour tout $1 \leq i < j \leq n$, alors $t(A) = 0$, c'est ce qui se produit si le tableau en entrée A est trié dans l'ordre croissant. En moyenne on peut supposer que $A[i]$ est strictement inférieur à $A[j]$ avec probabilité $1/2$, le sous tableau $A[1 \ldots j-1]$ étant trié, on a alors $t(A) = \sum_{j=2}^{j=n} \frac{j-1}{2} = \Theta(n^2)$. Asymptotiquement, le même nombre *quadratique* d'opérations est effectué dans le pire cas et dans le cas moyen, mais dans le meilleur cas, ce nombre est *linéaire*.

1.2.1 Paramètres de complexité

Logarithme et décomposition de base

Soit $d \geq 2$ un entier. Tout entier n admet une unique décomposition sous la forme $n = \sum_{j \geq 0} a_j d^j$, $a_j \in \{0, 1, \ldots, d-1\}$.

```
public class Decomp {
  static void Decom (int d, int n){
    int j=0;
    while(n>0){
      System.out.println("a" + j++ + "=" + n%d);
      n=n/d;   // floor rounding
    }
  }
  public static void main(String[] args){
    Decom(3,78);
  }
}
```

```
a0=0
a1=2
a2=2
a3=2
```

Pour tout $d \geq 2$. Si $n < d$ l'algorithme renvoie bien la décomposition de n et elle est unique. Soit $n \geq d$ le plus petit entier contre-exemple. On a $n = qd + a_0$ avec $q = \sum_{j \geq 0} b_j d^j > 0$ (car $q < n$, le plus petit contre-exemple). Donc $n = \sum_{j \geq 0} a_j d^j$ avec $a_j = b_{j-1}$ pour tout $j \geq 1$. De plus la décomposition de n est unique sinon $n = a_0 + \sum_{j \geq 1} c_j d^j$ et $q = \sum_{j \geq 1} c_j d^{j-1}$ aurait une autre décomposition (contradiction car $q < n$).

Pour tout réel $d > 1$, la fonction logarithme en base d, notée $\log_d(\cdot)$, associe à tout $x \in \mathbb{R}^+$, le réel z tel que $x = d^z$. Si $xy = d^{z_x} d^{z_y}$ et $xy = d^z$, alors $z = z_x + z_y$; donc $\log_d x + \log_d y = \log_d xy$ pour tout $x, y > 0$. Cela implique $\log_d x^k = k \log_d x$ pour tout $k \in \mathbb{N}$. On a $x/y = d^{z_x} d^{-z_y} = d^{z_x - z_y}$ implique $\log_d x - \log_d y = \log_d(x/y)$. Si $x = a^p = b^q$ et $b = a^r$ alors $a^p = a^{qr}$, d'où $p = qr$, donc

$$\log_a x = \log_a b \times \log_b x \quad \text{pour tout } a, b > 1 \text{ et tout } x > 0.$$

D'où l'existence, pour tout $a, b > 1$, d'une constante $c := \log_a b$, telle que $\log_a x = c \log_b x$; et donc $\log_a x = \Theta(\log_b x)$.

Pour tout entier n, il existe un unique entier k tel que $k \leq \lfloor \log_d n \rfloor < k+1$ c'est-à-dire $d^k \leq n < d^{k+1}$. Le nombre de bits codant l'entier n en base d est $\lfloor \log_d n \rfloor + 1$ en base $d \geq 2$. (Preuve : Clairement $a_k > 0$ et $a_j = 0$ pour tout $j > k$.)

Notez que $(\sum_{j=0}^{k} a_j d^j)^p = \ldots + a_k d^{pk}$, donc $\lfloor \log_d a^p \rfloor = p \lfloor \log_d a \rfloor$. De même $\sum_{j=0}^{k} a_j d^j \times \sum_{j=0}^{l} b_j d^j = \ldots + a_k b_l d^{k+l}$ avec $a_k b_l < d^2$. Donc $\lfloor \log_d (ab) \rfloor = \lfloor \log_d a \rfloor + \lfloor \log_d b \rfloor$ plus 1 ou non.

Par défaut $d = 10$ dans le sens que $\log(x) = \log_{10}(x)$ mais en algorithmique on utilise principalement le logarithme en base 2 noté $\log_2 x = \lg x$. De manière générale $\log_d x = \frac{\ln x}{\ln d}$ où $\ln(\cdot)$ est le logarithme néperien, mais la relation importante est $\log_d d^k = k$.

Par exemple, $\lg 8 = 3$ et $\log_3 9 = 2$, et

$n =$	16	32	64	128	256	512	1024	2048	4096	8192	16384
$\lg n =$	4	5	6	7	8	9	10	11	12	13	14

L'ordre de grandeur du lg est donné par l'approximation $\lg 10^{3k} \simeq 10k$. Ainsi le lg d'un milliard est de l'ordre de 30, et évident le $\log_{10}$ d'un milliard est 9. Pour les ordres de grandeur de l'algorithmique il est suffisant d'assimiler tout logarithme au logarithme en base 10, c'est-à-dire que l'on peut considérer que le log d'un entier est son nombre de digits (ou chiffres $0, 1, \ldots, 9$ en base 10). En fait on manipule des algorithmes de complexité logarithmique depuis la petite enfance puisque par exemple l'algorithme élémentaire de l'addition d'un entier à un entier n est en $O(\log n)$ c'est-à-dire qu'il nécessite moins de deux additions de deux chiffres par digits de n.

Paramètres en entrée : Tri dénombrement

La complexité dépend de ce que l'on suppose être constant, en général, par exemple, étant donné deux entiers a, b, on suppose que le test $a \leq b$ est constant, c'est-à-dire indépendant de la taille de a et b. Ainsi, la complexité de la plupart des algorithme de tri est exprimée uniquement en fonction du nombre n d'entiers à trier ; ce qui revient à considérer que la taille maximum d'un entier à trier est une constante du problème. L'algorithme suivant trie un tableau A de n éléments de $\{1, \ldots, k\}$ en $O(n+k)$, si l'on considère k comme une constante l'algorithme est linéaire, en $O(n)$.

Algorithm 2 TRI-DENOMBREMENT(A)

Soit A un tableau de n entiers $\leq k$ à trier.
Créer B et C deux tableaux de n et k entiers nuls.
for $j \leftarrow 1$ à n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 2$ à k **do**
  $C[i] \leftarrow C[i-1] + C[i]$
for $j \leftarrow 1$ à n **do**
  $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$
$A \leftarrow B$

(Dans la suite $j, j' \in \{1, \ldots, n\}$ et $i \in \{1, \ldots, k\}$.) La validité de **Algorithm 2** découle de l'invariant suivant : *à chaque passage dans la troisième boucle* **for**, *l'entier $C[i]$ est égal au nombre d'éléments j du tableau A tels que $A[j] \leq i$ et tel que j ne figure pas encore dans B.* (On dit que j figure dans B si l'on a déjà fait $B[j'] \leftarrow A[j]$ pour un j'.) En effet, cet invariant implique que dans la troisième boucle **for**, juste avant de faire figurer l'élément j de A dans B par $B[j'] \leftarrow A[j]$ avec $j' := C[A[j]]$, il y a j' éléments (y compris $A[j]$) à faire figurer dans B dont la valeur est $\leq A[j]$; ainsi j est à la bonne place dans B qui sera égal au tableau A trié juste avant la dernière ligne $A \leftarrow B$. Pour prouver l'invariant, il est facile de voir d'abord que, à la sortie de la première boucle **for**, $C[i] = |\{j : A[j] = i\}|$. Remarquons que l'on a aussi $C[1] = |\{j : A[j] \leq 1\}|$ et $C[2] = |\{j :$

$A[j] = 2\}|$. Comme $|\{j : A[j] \leq i\}| = |\{j : A[j] \leq i-1\}| + |\{j : A[j] = i\}|$, il s'ensuit par induction sur i que $C[i] = |\{j : A[j] \leq i\}|$ pour tout i après la deuxième boucle **for**.

Invariant de la troisième boucle **for :** $C[i] = |\{j : A[j] \leq i\}| - |\{j : B[j] = i\}|$

A l'entrée de la troisième boucle aucun j ne figure dans B, l'invariant est donc vrai avant la troisième boucle. A chaque parcours de la troisième boucle, l'élément j de A figure dans B après la première instruction, puis, à l'instruction suivante, on décrémente $C[A[j]]$; l'invariant est ainsi conservé ; fin de la preuve.

Remarquons que dans la troisième boucle **for** si on décrémente j le tri devient *stable* c'est-à-dire que pour tous $j < j'$ tels que $A[j] = A[j']$ avant l'algorithme, on aura à la dernière ligne, $B[k] = A[j]$ et $B[k'] = A[j']$ avec $k < k'$.

L'espace mémoire (en bits) requit pour faire fonctionner **Algorithm 2** est la complexité sont $O(n + k)$. Le nombre de bits codant l'entier k est $O(\lg k) := t$, donc la taille requise en mémoire pour stocker le tableau qui constitue l'entrée de **Algorithm 2** est $O(n + 2^t)$. L'espace mémoire pour faire fonctionner l'algorithme et sa complexité sont $\Omega(2^t)$, exponentiel par rapport à t, la taille de l'entrée (si on code l'entrée en binaire).

Paramètres en sortie

La complexité d'un algorithme dépend aussi du paramètre de sortie.

Considérons un algorithme qui calcule $\sqrt{2}$: $\sqrt{2}$ n'est pas rationnel. En effet, suppons qu'il existe deux entiers non nuls p et q tels que $\sqrt{2} = \frac{p}{q}$, et choissons un tel couple avec p le plus petit possible. Remarquons que si $p = 2n + 1$ avec $n \in \mathbb{N}$ alors $p^2 = 4n^2 + 4n + 1 = 2n' + 1$ avec $n' \in \mathbb{N}$. Ainsi p^2 n'est divisible par 2 que si p l'est aussi. Puisque $p^2 = 2q^2$ alors $p = 2n$. D'où $q^2 = 4n^2/2 = 2n^2$. Mais alors q est aussi divisible par 2 ce qui contredit la minimalité de p.

```
public class Heron {
  final static double EPSILON=1.E-9;
  static int compt=0;
  static int heron(double rac){
    compt=0;
    double a=rac;
    while(Math.abs(a-Math.sqrt(rac)) > EPSILON){
      ++compt;
      a=(rac/a + a)/2.0;
    }
    return compt;
  }
  public static void main(String[] args){
    long start,finish;
      int test=0,max=1;
      start = System.currentTimeMillis();
      for(int i = 2; i<1.E9 ; i++){
        test=heron(i);
        if( max < test)
          max=test;
      }
      finish = System.currentTimeMillis();
      System.out.println("Total time= " + (finish-start) + "ms\t Max steps=" + test);
  }
}
```

donne

```
Total time= 316563ms  Max steps=19
```

Pour le calcul de racine de 2, la sortie est donc de taille infinie, la complexité du calcul dépend donc de la précision demandée. La suite $u_{n+1} = \frac{u_n}{2} + \frac{1}{u_n}$ converge vers $\sqrt{2}$ (voir géométrie grec ou méthode de Newton). Avec $u_0 = 1$ on obtient $u_5 = 1.41421356237309504880168$.

La limite principale du concept de complexité est numérique. Pour les méthodes numériques en général le concept de convergence est plus pertinent que celui de complexité.

1.3 Expérimentations sur le temps de calcul

1.3.1 Coefficient binômial

En pratique, $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$ n'ont de sens que si n peut atteindre de grandes valeurs, et ainsi la pertinence du concept de complexité est liée à celle de l'étude du comportement asymptotique de l'algorithme. On peut par-exemple calculer en $\Theta(n)$ ou en $\Theta(n^2)$ le coefficient binomial $\binom{n}{p} = \frac{n!}{p!(n-p)!}$ qui atteind son maximum pour $p = \lfloor n/2 \rfloor$ ou $\lceil n/2 \rceil$ mais dans la pratique on est plus limité par la taille mémoire que par le temps d'exécution. En pratique, on peut instantanément calculer un coefficient binomial $\binom{n}{p}$ inférieur à la taille maximum d'un `double`, soit $\binom{1029}{514}$.

```
public class Binomial {
  public static double BinomialRec(int n, int p){
    if(p==n || p==0)
      return 1.0;
    else
      return binomialRec(n-1,p-1)+binomialRec(n-1,p);
  }
public static double BinomialIt(int n, int p){
  if ( (n-p) < p ) p=n-p;
  // Allocate tri1 and tri2
  double[][] tri1 = new double [p+1][];
  double[][] tri2 = new double [p+1][];
  for(int i = 0 ; i < p+1 ; i++){
      tri1[i] = new double[i+1];
      tri2[i] = new double[i+1];
  }
  // Construct tri1
  for(int i = 0 ; i < p+1; i++)
  tri1[i][0] = tri1[i][i] = 1;
    for(int i = 2 ; i < p+1; i++)
      for(int j = 1 ; j <= i -1 ; j++)
       tri1[i][j] = tri1[i-1][j-1] + tri1[i-1][j];
  // Intitialization tri2[p][]
    if (n-2*p-1 == -1)  // n is even and p=n/2
       for(int j = p ; j >= 0 ; j--)
      tri2[p][j]=tri1[p][p-j];
    if (n-2*p-1 == 0){
     tri2[p][p]=1;
     for(int j = p-1 ; j >= 0 ; j--)
      tri2[p][j] = tri1[p][p-j-1] + tri1[p][p-j];
    }
     if (n-2*p-1 >= 1){
     double[][] rect = new double [n-2*p-1][p+1];
     for(int i = 0 ; i < n-2*p-1; i++)
       rect[i][0]=1;
     for(int j = 1 ; j < p+1 ; j++)
       rect[0][j] = tri1[p][j-1] + tri1[p][j];
```

```
     for(int i = 1 ; i < n-2*p-1; i++)
       for(int j = 1 ; j < p+1; j++)
        rect[i][j] = rect[i-1][j-1] + rect[i-1][j];
     tri2[p][p]=1;
     for(int j = p-1 ; j >= 0 ; j--)
          tri2[p][j] = rect[n-2*p-2][p-j-1] + rect[n-2*p-2][p-j];
    }
    // Construct tri2
    for(int i = p-1 ; i >= 0; i--)
        for(int j = i ; j >= 0; j--)
          tri2[i][j] = tri2[i+1][j+1] + tri2[i+1][j];
  return tri2[0][0];
  }

static double BinomialDir(int n, int k) {
  return Math.floor(0.5 + Math.exp(logFactorial(n) - logFactorial(k) - logFactorial(n-k)));
}
  static double logFactorial(int n) {
  double ans = 0.0;
  for (int i = 1; i <= n; i++)
    ans += Math.log(i);
  return ans;
}

  public static double TriPasc(int n){
    double[][] mat = new double [n+1][];
    for(int i = 0 ; i < n+1 ; i++)
      mat[i] = new double[i+1];
    for(int i = 0 ; i < n+1; i++)
      mat[i][0] = mat[i][i] = 1;
    for(int i = 2 ; i < n+1; i++)
      for(int j = 1 ; j <= i -1 ; j++)
        mat[i][j] = mat[i-1][j-1] + mat[i-1][j];
  }
  public static double fact(int n)
  { double res = 1.0;
    for(int i = 1; i <= n ; i++)
      res *=i;
    return res;
  }
  public static double arrangement(int n, int p)
  { double res = 1.0;
    for(int i = n ; i > n - p ; i--)
      res *= i;
    return res;
  }

  public static double binomial(int n, int p)
  { return arrangement(n,p)/fact(p);}

  public static void main(String[] args){
    final int n = 268;
    double[][] mat = new double [n+1][];
    for(int i = 0 ; i < n+1 ; i++)
```

```
      mat[i] = new double[i+1];
    for(int i = 0 ; i < n+1; i++)
      mat[i][0] = mat[i][i] = 1;
    for(int i = 2 ; i < n+1; i++)
      for(int j = 1 ; j <= i -1 ; j++)
        mat[i][j] = mat[i-1][j-1] + mat[i-1][j];
    System.out.println("Triangle Pascal = " + mat[n][n/2]);
    System.out.println("Binomial = " + binomial(n,n/2));
  }
}
```

La validité de `Triangle Pascal` découle, par induction sur n, de :

$$\begin{aligned}\binom{n}{p} &:= \Big|\big\{A \subseteq \{1,\dots,n\} : |A| = p\big\}\Big| \\ &= \Big|\big\{A \subseteq \{2,\dots,n\} : |A| = p\big\}\Big| + \Big|\big\{A \subseteq \{2,\dots,n\} : |A| = p-1\big\}\Big| \\ &= \binom{n-1}{p} + \binom{n-1}{p-1}\end{aligned}$$

Puisque $0! = 1$ on a $\binom{n}{p} = \frac{n!}{(n-p)!p!}$ pour $p = 0$ et $p = n$, et par induction :

$$\begin{aligned}\frac{(n-1)!}{(n-1-p)!p!} + \frac{(n-1)!}{(n-p)!(p-1)!} &= \frac{(n-p)\,(n-1)!}{(n-p)!p!} + \frac{p\,(n-1)!}{(n-p)!p!} \\ &= \frac{n!}{(n-p)!p!}\end{aligned}$$

La complexité de `binomial(int n, int p)` est $\Theta(n)$ et le calcul du triangle de Pascal `double[][] mat` est en $\Theta(n^2)$. Avec `n = 268;` on a instantanément :

```
Triangle Pascal = 2.309438374333814E79
Binomial        = 2.309438374333815 7E79
```

La valeur arrondie donnée par la méthode en $\Theta(n^2)$ est correcte tandis que la méthode en $\Theta(n)$ fait une légère erreur. Avec `n = 269;` on a :

```
Triangle Pascal = 4.6017697977466374E79
Binomial        = Infinity
```

Pour pouvoir poursuivre avec une méthode en $\Theta(n)$ il faut modifier `binomial` pour :

```
static double logFactorial(int n) {
  double ans = 0.0;
  for (int i = 1; i <= n; i++)
    ans += Math.log(i);
  return ans;
}
static double binomial(int n, int k) {
  return
Math.floor(0.5 + Math.exp(logFactorial(n) - logFactorial(k) - logFactorial(n-k)));
}
```

On a utilisé le fait que $\lfloor 0.5 + x \rfloor$ est l'entier le plus proche [1] de x, que $\binom{n}{p} = \mathrm{e}^{\ln n! - \ln p! - \ln(n-p)!}$, et que $\ln n! = \sum_{k=1}^{k=n} \ln k$. Ainsi pour `n = 1029;` on a instantanément :

1. En fait l'entier n tel que $|x - n| < 0.5$ si il existe, et sinon $\lceil x \rceil$.

```
Triangle Pascal = 1.4298206864989042E308
Binomial        = 1.4298206864981146E308
```

(notez la légère différence). Et avec `n = 1030;` on a :

```
Triangle Pascal = Infinity
Binomial        = Infinity
```

```
For n= 25 Binomial Rec. took  31ms
For n= 26 Binomial Rec. took  47ms
For n= 27 Binomial Rec. took  140ms
For n= 28 Binomial Rec. took  203ms
For n= 29 Binomial Rec. took  500ms
For n= 30 Binomial Rec. took  782ms
For n= 31 Binomial Rec. took  1937ms
For n= 32 Binomial Rec. took  3078ms
For n= 33 Binomial Rec. took  7485ms
For n= 34 Binomial Rec. took  11906ms
For n= 35 Binomial Rec. took  29094ms
For n= 36 Binomial Rec. took  46562ms
For n= 37 Binomial Rec. took  113344ms
For n= 38 Binomial Rec. took  180562ms
For n= 39 Binomial Rec. took  441797ms
For n= 40 Binomial Rec. took  707563ms
For n= 41 Binomial Rec. took  1727937ms

Binomial with Log  = 1.4298206864981146E308   (n=1029)
Binomial It  = 1.4298206864989042E308   (n=1029)
Total time  0ms
```

Soient n et $p_1, \ldots, p_k$ des entiers tels que $\sum_{i=1}^{k} p_i = n$, alors on note $\binom{n}{p_1,\ldots,p_k}$ le nombre de partitions $(V_1, \ldots, V_k)$ de $\{1, \ldots, n\}$ avec $|V_i| = p_i$. On peut montrer que

$$\binom{n}{p_1, \ldots, p_k} = \binom{n-1}{p_1 - 1, \ldots, p_k} + \binom{n-1}{p_1, p_2 - 1, \ldots, p_k} + \ldots \binom{n-1}{p_1, \ldots, p_k - 1}$$

et que

$$\binom{n}{p_1, \ldots, p_k} = \frac{n!}{p_1! \ldots p_k!}$$

1.3.2 Termes de Fibonacci

Souvenons nous de l'exemple de la suite de Fibonacci, pour tout entier $n \geq 2$, on définit $F_n = F_{n-2} + F_{n-1}$ avec au départ $F_i = i$ pour $i = 0, 1$.

L'implémentation en Java suivante :

```
public class Fibonacci {
  static int FiboRec (int n){
    if (n<2) return n;
    else return FiboRec(n-1)+FiboRec(n-2);
  }
  static double FiboIt (int n){
    if (n<2) return n;
    else{
      double x=0,y=1,z;
      for(int i=2;i<=n;i++){
```

```
        z=x+y; x=y; y=z;
      }
    return z;
    }
  }
  static double FiboDir (int n){
    double res;
    res = (Math.pow(phi,n)+Math.pow(phic,n))/Math.sqrt(5);
    return Math.floor(res + 0.5);
  }
  static final double phi=(1+Math.sqrt(5))/2, phic=(1-Math.sqrt(5))/2;
  static long start,finish;
  static double MaxDistGoldRat(int from, int to){
    double max=0;
    start = System.currentTimeMillis();
    FiboRec(from-1);
    finish = System.currentTimeMillis();
    double time=finish-start;
    for(int n=from; n<=to;n++){
      start = System.currentTimeMillis();
      FiboRec(n);
      finish = System.currentTimeMillis();
      if (max < Math.abs( ((finish-start)/time) - phi) )
        max=Math.abs( ((finish-start)/time) - phi);
      time=finish-start;
    }
    return max;
  }
  public static void main(String[] args){
    System.out.println("Max dist with Golden Ratio = " + MaxDistGoldRat(45,50) );
  }
}
```

donne :

```
Max dist with Golden Ratio = 0.01391776598960437
```

On a utilisé la procédure récursive suivante qui prend en paramètre d'entrée un entier n et donne, en paramètre de sortie, l'entier égal à la valeur de F_n :

Algorithm 3 FIBONACCI-REC(n)

if $n < 2$ **then**
  return n
else
  return FIBONACCI-REC($n-1$)+FIBONACCI-REC($n-2$)

On a aussi procédé itérativement :

Et aussi directement avec **Algorithm 5** dont la validité est un résultat de mathématique classique : ϕ est le nombre d'or, c'est-à-dire la solution positive de $x^2 = x+1$, et $\overline{\phi}$ est l'autre solution (le conjugué). Ainsi $x = \phi$ et $x = \overline{\phi}$ satisfont $x^n = x^{n-1} + x^{n-2}$ pour tout $n \geq 2$, et donc $x_n = a\phi^n + b\overline{\phi}^n$ satisfait $x_n = x_{n-1} + x_{n-2}$ pour tout $a, b \in \mathbb{R}$. Pour avoir $x_0 = 0$, on a $b = -a$, et pour avoir $x_1 = 1$ on a $a = \frac{1}{\phi - \overline{\phi}}$. D'où, l'unicité étant claire, $F_n = \frac{1}{\sqrt{5}}(\phi^n - \overline{\phi}^n)$.

La validité de **Algorithm 3** découle du raisonnement classique *par récurrence (ou par induction)* : D'une part, trivialement, l'algorithme marche avec tout entier $n < 2$, d'autre part, pour tout $n \geq 2$, si il marche pour tous les entiers strictement inférieurs à n alors il marche pour n.

Algorithm 4 FIBONACCI-IT(n)

```
if n < 2 then
  return n
else
  a ← 0 ; b ← 1
  for i ← 2 à n do
    c ← a + b ; a ← b ; b ← c
return c
```

Algorithm 5 FIBONACCI-DIRECT(n)

```
φ ← (1+√5)/2
φ̄ ← (1−√5)/2
return (1/√5)(φ^n − φ̄^n)
```

Trivialement **Algorithm 4** marche avec tout entier $n < 2$. Pour $n \geq 2$ on rentre dans la boucle **for** initialisée avec $i = 2$, $a = F_{i-2}$ et $b = F_{i-1}$. Après la première instruction $c = F_i$. Les deux autres instructions assurent que l'on a l'*invariant* suivant : *à l'entrée de la boucle* **for** *on a* $a = F_{i-2}$ *et* $b = F_{i-1}$. Ainsi à la sortie de la boucle **for** on a $i = n$ et donc $c = F_n$, d'où la validité [2].

La description de **Algorithm 5** n'est que partielle car le calcul de mise à la puissance n n'est pas explicité, de même que la manipulation de $\sqrt{5}$.

Dans **Algorithm 5**, on a besoin de la fonction puissance et aussi de $\sqrt{5}$. On verra qu'il existe un algorithme logarithmique pour la fonction puissance, ce qui pourrait améliorer l'approche linéaire. Mais quelque soit la valeur stockée de $\sqrt{5}$ sa précision pourrait ête insuffisante car $\sqrt{5}$ n'est pas rationnel. En effet, suppons qu'il existe deux entiers non nuls p et q tels que $\sqrt{5} = \frac{p}{q}$, et choissons un tel couple avec p le plus petit possible. Remarquons que si $p = 5n + r$ avec $n \in \mathbb{N}$ et $r \in \{1, 2, 3, 4\}$ alors $p^2 = 25n^2 + 10nr + r^2 = 5n' + r'$ avec $n' \in \mathbb{N}$ et $r' \in \{1, 4\}$ (car $9 = 5 + 4$ et $16 = 15 + 1$). Ainsi p^2 n'est divisible par 5 que si p l'est aussi. On a $p^2 = 5q^2$ implique que $p = 5n$. D'où $q^2 = 25n^2/5 = 5n^2$. Mais alors q est aussi divisible par 5 ce qui contredit la minimalité de p. Une manière de se débarasser du problème de l'approximation sera de calculer $\phi^n = 1/2^n(1 + \sqrt{5})^n$ via la formule du binôme de Newton :

$$(a+b)^n = \sum_{i=0}^{i=n} \binom{n}{i} a^i b^{n-i},$$

mais on retomberait sur du linéaire. Par induction :

$$\begin{aligned}
(a+b)^{n+1} &= (a+b)\sum_{i=0}^{i=n} \binom{n}{i} a^i b^{n-i} \\
&= \sum_{i=0}^{i=n} \binom{n}{i} a^{i+1} b^{n-i} + \sum_{i=0}^{i=n} \binom{n}{i} a^i b^{n-i+1} \\
&= \sum_{i=1}^{i=n+1} \binom{n}{i-1} a^i b^{n-i+1} + \sum_{i=0}^{i=n} \binom{n}{i} a^i b^{n-i+1} \\
&= a^0 b^{n+1} + \sum_{i=0}^{i=n} \left(\binom{n}{i-1} + \binom{n}{i} \right) a^i b^{n-i+1} + a^{n+1} b^0 \\
&= \sum_{i=0}^{i=n+1} \binom{n+1}{i} a^i b^{n+1-i}
\end{aligned}$$

2. Au niveau machine i sera en fait incrémenté une fois de plus.

Soient $A_1(n)$ et $A_2(n)$ les nombres d'additions effectuées par **Algorithm 3** et **Algorithm 4** respectivement.

n	0	1	2	3	4	5	6
F_n	0	1	1	2	3	5	8
$A_1(n)$	0	0	1	2	4	7	12
$A_2(n)$	0	0	1	2	3	4	5

Clairement, $A_1(n) = A_1(n-1) + A_1(n-2) + 1$ et $A_2(n) = n - 1$, pour $n \geq 2$. Ainsi, d'une part, $A_1(n) = F_{n+1} - 1$ pour $n = 0, 1$, et d'autre part, par induction, cette égalité reste vraie pour tout $n \geq 2$, car alors $A_1(n+1) = (F_{n+1} - 1) + (F_n - 1) + 1 = F_{n+2} - 1$. Donc $A_1(n) \geq F_n + 1$ pour $n \geq 4$. Puisque $1 < \sqrt{5} < 3$, on a $|\overline{\phi}| < 1$ et donc $F_n \geq \frac{1}{\sqrt{5}}(\phi^n - 1)$, donc de plus on a $F_n + 1 \geq \frac{\phi^n}{\sqrt{5}}$. La fonction $A_1(n)$ appartient à l'ensemble des fonctions $f(n)$ suivant :

$$\Omega(\phi^n) = \{f(n) : 0 \leq c\phi^n \leq f(n), \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

En effet, il suffit de prendre $c = \frac{1}{\sqrt{5}}$ et $n_0 = 4$. En notation asymptotique, on écrit $A_1(n) = \Omega(\phi^n)$. Puisque $\phi \geq 1.618033$, on a aussi numériquement $A_1(n) = \Omega(1.6^n)$. Notez qu'il est possible d'avoir $f \neq g$ et $\Omega(f) = \Omega(g)$.

Par-ailleurs, il est clair que $A_2(n)$ appartient à l'ensemble des fonctions $f(n)$ suivant :

$$O(n) = \{f(n) : 0 \leq f(n) \leq cn, \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

En effet, il suffit de prendre $c = 1$ et $n_0 = 0$ (car $A_2(0) = A_2(1) = 0$). En notation asymptotique, on écrit $A_2(n) = O(n)$. Avec $c = 1/2$ et $n_0 = 2$ il apparait que l'on a aussi $A_2(n) = \Omega(n)$, et donc $A_2(n)$ appartient à l'ensemble des fonctions $f(n)$ suivant :

$$\Theta(n) = \{f(n) : 0 \leq c_1 n \leq f(n) \leq c_2 n, \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0\}.$$

En notation asymptotique, on écrit $A_2(n) = \Theta(n)$.

```
For n=30 Fibonacci Dir. It. Rec. took  0ms 0ms      31ms
For n=31 Fibonacci Dir. It. Rec. took  0ms 0ms      16ms
For n=32 Fibonacci Dir. It. Rec. took  0ms 0ms      47ms
For n=33 Fibonacci Dir. It. Rec. took  0ms 0ms      78ms
For n=34 Fibonacci Dir. It. Rec. took  0ms 0ms     109ms
For n=35 Fibonacci Dir. It. Rec. took  0ms 0ms     188ms
For n=36 Fibonacci Dir. It. Rec. took  0ms 0ms     297ms
For n=37 Fibonacci Dir. It. Rec. took  0ms 0ms     484ms
For n=38 Fibonacci Dir. It. Rec. took  0ms 0ms     766ms
For n=39 Fibonacci Dir. It. Rec. took  0ms 0ms    1249ms
For n=40 Fibonacci Dir. It. Rec. took  0ms 0ms    2016ms
For n=41 Fibonacci Dir. It. Rec. took  0ms 0ms    3266ms
For n=42 Fibonacci Dir. It. Rec. took  0ms 0ms    5296ms
For n=43 Fibonacci Dir. It. Rec. took  0ms 0ms    8563ms
For n=44 Fibonacci Dir. It. Rec. took  0ms 0ms   13827ms
For n=45 Fibonacci Dir. It. Rec. took  0ms 0ms   22453ms
For n=46 Fibonacci Dir. It. Rec. took  0ms 0ms   36280ms
For n=47 Fibonacci Dir. It. Rec. took  0ms 0ms   58670ms
For n=48 Fibonacci Dir. It. Rec. took  0ms 0ms   94762ms
For n=49 Fibonacci Dir. It. Rec. took  0ms 0ms  153620ms
For n=50 Fibonacci Dir. It. Rec. took  0ms 0ms  250024ms
For n=51 Fibonacci Dir. It. Rec. took  0ms 0ms  404737ms
For n=52 Fibonacci Dir. It. Rec. took  0ms 0ms  653968ms
For n=53 Fibonacci Dir. It. Rec. took  0ms 0ms 1056680ms
Error = 7
```

```
Fibonacci 1470  Dir. 7.28360130920199E306      It. 7.283601309201629E306
Fibonacci 1471  Dir. 1.1785114478792054E307    It. 1.1785114478791467E307
Fibonacci 1472  Dir. 1.9068715787994046E307    It. 1.9068715787993096E307
Fibonacci 1473  Dir. 3.08538302667861E307      It. 3.085383026678456E307
Fibonacci 1474  Dir. 4.992254605478015E307     It. 4.992254605477766E307
Fibonacci 1475  Dir. Infinity                  It. 8.077637632156222E307
Fibonacci 1476  Dir. Infinity                  It. 1.3069892237633987E308
Fibonacci 1477  Dir. Infinity                  It. Infinity
Total took 0ms
```

La progression du temps de calcul de l'implémentation Java `FiboRec` est bien du type $\Theta(\phi^n)$. Les temps de calculs par exemple pour $n = 46$ et 47 sont $58670 = 36280\tilde{\phi}$ avec $|\tilde{\phi} - \phi| < 0.01$. Les temps théoriques supportent la comparaison avec les temps expérimentaux :

```
Theoretical time for n=46   36280ms
Theoretical time for n=47   58702ms
Theoretical time for n=48   94982ms
Theoretical time for n=49  153684ms
Theoretical time for n=50  248666ms
Theoretical time for n=51  402351ms
Theoretical time for n=52  651018ms
Theoretical time for n=53 1053369ms
```

On peut se demander quand faudrait-il avoir commencé le calcul récursif de F_n pour en venir à bout ? Pour $n = 75$: l'année dernière. Pour $n = 80$: il y a dix ans. Pour $n = 85$: il y a cent ans. Pour $n = 90$: il y a mille ans. En fait pour $n = 123$ le temps théorique est supérieur à 13,7 milliards d'années, l'estimation de l'âge de l'univers. Avec la méthode linéaire on a calculé en `0ms` pour $n = 1474$:

```
Theoretical time for n=1474              9.863193545216687E302ms
```

A peu près 10^{300} secondes, soit $\frac{10^{300}}{3600\times24\times365} = 5.10^{292}$ années.

Le calcul par la méthode linéaire est limité, non par le temps, mais par le type `double` puisque sa taille maximum est de l'ordre de $1.8 \cdot 10^{308}$ (précisément $(2 - 2^{-52}) \cdot 2^{1023}$, notez qu'à cet ordre de grandeur on atteint les limites de l'approximation $2^{3k} \simeq 10^k$ basée sur $\log 2 \simeq 3.010$ puisque $10^{1023/3} = 10^{341}$). C'est aussi ce qui explique la différence des valeurs à partir d'une certaine taille de n (les 7 erreurs constatées sont pour $n > 46$).

En règle général, pour une implémentation raisonnable permettant d'utiliser de grands paramètres la théorie reflète la réalité. Nous comparerons les temps théoriques et pratiques d'autres algorithmes, en particulier nous verrons le rôle de l'implémentation (avec l'algorithme de Strassen).

Chapitre 2

Outils d'analyse algorithmique

L'analyse d'un algorithme consiste en deux points : après la preuve de sa validité, l'évaluation de sa complexité. Voici des outils sous forme de résultats fondamentaux pour le calcul de complexité et de concepts théoriques puissants pour les preuves de validité.

2.1 Puissance et logarithme

Une fonction $f(x)$ est une *fonction puissance* si il existe une constante b telle que $f(x) = b^x$. On note $x^{-q} = \frac{1}{x^q}$ et $x^{1/q} := y$ tel que $y^q = x$ pour tout entier q ; ce qui permet de définir x^y pour tout réel y (car $\mathbb{Q}$ est dense dans $\mathbb{R}$, cf cours d'analyse). Une fonction $f(x)$ est puissance si elle vérifie :

$$f(x+y) = f(x)f(y)$$

ce qui implique :
(1) $f(0) = 1$;
(2) $f(x+y) = f(x)f(y)$;
(3) $f(x-y) = f(x)/f(y)$.
En effet, $f(x) = f(x+0) = f(x)f(0)$ donc (1). (2) trivial. $1 = f(x-x) = f(x)f(-x)$ donc $f(-x) = 1/f(x)$ d'où (3).

Par (1) et (3) on a que $f(x)$ est injective, car si $f(x) = f(y)$, alors $1 = f(x)/f(y) = f(x-y)$ donc $x - y = 0$. Soit $b := f(1)$. (2) implique $f(qx) = f(x)^q$ pour tout entier q. D'où (avec $x = 1$), $f(q) = b^q$ pour tout entier q. Si $x = 1/q$ pour q entier, alors $f(qx) = f(1) = b = f(x)^q$ donc $f(x) := y$ tel que $y^q = b$, en d'autres termes $f(x) = b^{1/q} = b^x$. Donc $f(x) = b^x$ pour tout rationnel x, que l'on peut étendre à tout réel x.

Une fonction $f(x)$ est une *fonction logarithme* si il existe une constante b telle que $f(x) := y$ tel que $b^y = x$; la fonction $f(x)$ est alors notée $\log_b(x)$. Une fonction $f(x)$ est logarithme si elle vérifie :

$$f(xy) = f(x) + f(y)$$

ce qui implique :
(i) $f(1) = 0$;
(ii) $f(xy) = f(x) + f(y)$;
(iii) $f(x/y) = f(x) - f(y)$.
En effet, $f(x) = f(1.x) = f(1) + f(x)$ donc (i). (ii) trivial. $0 = f(x.(1/x)) = f(x) + f(1/x)$ donc $f(1/x) = -f(x)$ d'où (iii).

Par (i) et (iii) on a que $f(x)$ est injective, car si $f(x) = f(y)$, alors $0 = f(x) - f(y) = f(x/y)$ d'où $x/y = 1$. Donc l'inverse $b := f^{-1}(1)$ est bien défini. (ii) implique que $f(x^p) = pf(x)$ pour tout entier p. On a $qf(x^{1/q}) = f(x)$ donc $f(x^{1/q}) = \frac{1}{q}f(x)$. D'où $f(x^q) = qf(x)$ pour tout rationnel q, que l'on peut étendre à tout réel. Soit un réel x, puisqu'une fonction puissance est injective, y tel que $b^y = x$ est bien défini de manière unique. On a alors $f(x) = f(b^y) = yf(b) = y$, d'où $f(x)$ est la fonction logarithme $\log_b(x)$.

Rappel :

$$\log_b x = \log_b a \times \log_a x$$

2.2 Complexité des algorithmes récursifs

2.2.1 Analyse récursive

Fibonacci et coefficient binômial

Le paradigme "diviser pour régner" ne recouvre pas tous les algorithmes récursifs. Analysons une troisième méthode, récursive, pour le calcul de $\binom{n}{p}$:

```
public static double binomialRec(int n, int p){
  if(p==n || p==0)
    return 1.0;
  else
    return binomialRec(n-1,p-1)+binomialRec(n-1,p);
}
```

Le temps d'exécution vérifie $T(n,p) = T(n-1,p-1) + T(n-1,p) + 1$. Le théorème ne peut être appliqué, mais on remarque que l'arrêt d'un appel récursif se solde par un retour de la valeur 1. Ainsi $\binom{n}{p}$ a été décomposé en $\binom{n}{p} = 1+1+\ldots+1$, et donc l'algoritme effectue $\binom{n}{p} - 1$ additions en tout. Son temps d'exécution est donc exponentiel en p, car :

$$\binom{n}{p} = \frac{n}{p}\frac{n-1}{p-1}\cdots\frac{n-p+1}{1} \geq \left(\frac{n}{p}\right)^p$$

Soit, pour $p = n/2$, une complexité $\Omega(1.41^n)$. On constate l'explosion combinatoire en pratique :

```
For n= 25 Binomial Rec. took       31ms
For n= 26 Binomial Rec. took       47ms
For n= 27 Binomial Rec. took      140ms
For n= 28 Binomial Rec. took      203ms
For n= 29 Binomial Rec. took      500ms
For n= 30 Binomial Rec. took      782ms
For n= 31 Binomial Rec. took     1937ms
For n= 32 Binomial Rec. took     3078ms
For n= 33 Binomial Rec. took     7485ms
For n= 34 Binomial Rec. took    11906ms
For n= 35 Binomial Rec. took    29094ms
For n= 36 Binomial Rec. took    46562ms
For n= 37 Binomial Rec. took   113344ms
For n= 38 Binomial Rec. took   180562ms
For n= 39 Binomial Rec. took   441797ms
For n= 40 Binomial Rec. took   707563ms
For n= 41 Binomial Rec. took  1727937ms
```

Cette argumentation est générale. Analysons de cette manière une version légèrement modifiée de l'algorithme récursif pour Fibonacci. Au lieu de calculer F_2 récursivement on retourne maintenant la valeur 1 pour F_1 et F_2. Ainsi, l'algorithme décompose F_n en $F_n = 1+1+\ldots+1$ en effectuant un total de $F_n - 1$ additions. La complexité est donc toujours $\Theta(\phi^n)$ mais en pratique l'exécution est plus rapide :

```
For n=46 Fibonacci Rec. modified took  10813ms
For n=47 Fibonacci Rec. modified took  17484ms
For n=48 Fibonacci Rec. modified took  28360ms
For n=49 Fibonacci Rec. modified took  45797ms
```

Une manière rapide de voir que F_n est exponentielle consiste à remarquer que

$$F_n \geq 2F_{n-2} \geq 2^k F_{n-2k} \geq 2^{\frac{n-1}{2}} = \Omega(1.41^n)$$

Origine historique : PGCD

```
public class pgcd {
  static double PcgdRec (double a, double b){
    if (b == 0)
      return a;
    else
      return PcgdRec(b, a % b);
  }
  static double PcgdIt (double a, double b){
    double r;
    while ((r = a % b) != 0){
      a = b;
      b = r;
    }
    return b;
  }
  public static void main(String[] args){
    long start;
    final double BIG=1.0E9;
    int times= (int) BIG;
    double a,b;
    Random generator = new Random(485741301);
    start = System.currentTimeMillis();
    while(times>=0){
      a = generator.nextDouble()*1.7976931348623157E308d;
      b = generator.nextDouble()*1.7976931348623157E308d;
      if (a>b) PcgdRec(a,b);
      else PcgdRec(b,a);
      times=times-1;
    }
    System.out.println(" PGCD" + " by rec. \t \t took "
      + (System.currentTimeMillis() - start) + "ms");
    start = System.currentTimeMillis();
    times= (int) BIG;
    while(times>=0){
    a = generator.nextDouble()*1.7976931348623157E308d;
    b = generator.nextDouble()*1.7976931348623157E308d;
    if (a>b) PcgdIt(a,b);
    else PcgdIt(b,a);
    times=times-1;
    }
    System.out.println(" PGCD" + " by it. \t \t took "
      + (System.currentTimeMillis() - start) + "ms");
  }
}
```

Il faut un peu plus de 5 minutes pour 10^9 calculs aléatoires de PGCD de la taille maximum d'un double on a :

```
PGCD by rec. took 468734ms
PGCD by it.  took 449579ms
```

Par induction sur a, la validité de l'algorithme découle du fait que $pgcd(a,b) = pgcd(b,r)$ si r est le reste de la division entière de a par b (avec la convention $pgcd(b,0) = b$), puisque $a = qb + r$ avec $r < b$ implique que d est un diviseur de b et r si et seulement si d est un diviseur de a et b.

En 1844, Lamé a montré que l'algorithme d'Euclide pour le calcul de $pgcd(a,b)$ avec $a > b$ est en $O(\lg b)$. On peut supposer que $pgcd(a,b) = 1$ car le nombre de divisions à effectuer est le même dans les calculs de $pgcd(a,b)$ et de $pgcd(a/d, b/d)$ avec $d = pgcd(a,b)$. Ainsi soient $b, \ldots, d_5, d_4, d_3, d_2, 1$ les diviseurs successifs, c'est-à-dire les valeurs de `b`, dans le déroulement de l'algorithme d'Euclide. On a $d_2 \geq 2$ et $d_{i+2} \geq d_{i+1} + d_i$. Donc $b = d_n$ avec $d_n \geq F_n = \Omega(\phi^n)$ (où F_n est le $n$ième terme de la suite de Fibonnacci et ϕ est le nombre d'or). D'où $n = O(\lg b)$.

(La démonstration originale est que $d_3 \geq 3$, $d_4 \geq 5$, $d_5 \geq 8$, $d_6 \geq 13$, $d_7 \geq 21$. Ce qui valide $d_{5k+1} > 10^k$ et $d_{5k+2} > 2.10^k$ pour $k = 1$. C'est vrai pour tout k par induction, car $d_{5k+3} > 3.10^k$, $d_{5k+4} > 5.10^k$, $d_{5(k+1)} > 8.10^k$ puis $d_{5(k+1)+1} > 13.10^k > 10^{k+1}$. Donc, si $b = d_{5k+t}$, c'est-à-dire si l'on effectue plus de $5k$ divisions, alors b a au-moins k digits (en base 10). D'où la complexité en $O(\lg b)$.)

2.2.2 Algorithmes "divide-and-conquer"

D'une manière générale, un algorithme conçu suivant le paradigme diviser pour régner a un temps d'éxécution $T(n)$ satisfaisant l'équation de récurrence

$$T(n) = aT(n/b) + f(n)$$

avec $T(1) = O(1)$ et $f(1) = 0$, et où n est la taille des entrées, a est le nombre de sous-problèmes, n/b est la taille des sous-problèmes, et $f(n)$ est le temps requis pour diviser et combiner.

Par-exemple, dans le tri fusion (**Algorithm 6**), sachant que `Fusionner` est en $\Theta(n)$, on a $T(n) = 2T(n/2) + \Theta(n)$ avec $T(1) = O(1)$, dont on verra que la solution est $T(n) = \Theta(n \lg n)$.

Algorithm 6 TRI-FUSION(A, i, k)

if $i < k$ **then**
  $j \leftarrow \lfloor \frac{i+k}{2} \rfloor$
  TRI-FUSION(A, i, j)
  TRI-FUSION$(A, j+1, k)$
  FUSIONNER (A, i, j, k)

On peut aussi calculer récursivement le produit $C = AB$ de deux matrices carrées A et B, En supposant que les matrices A, B sont de taille 2^k, on a :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

où X_{ij} $(X = A, B, C)$ est la sous-matrice carrée de X formée des 2^{k-1} premières (resp. dernières) lignes si $i = 1$ (resp. si $i = 2$) et des 2^{k-1} premières (resp. dernières) colonnes si $j = 1$ (resp. si $j = 2$). Chaque calcul récursif d'un des 8 produits $A_{ij}B_{kl}$ $i, j, k, l \in \{1, 2\}$ composants C de matrices carrées $n/2 \times n/2$ prends un temps $T(n/2)$ si $T(n)$ est le temps de multiplication de deux matrices carrées $n \times n$. Puisque l'addition de deux matrices carrées $n \times n$ est $\Theta(n^2)$, alors $T(n)$ satisfait $T(n) = 8T(n/2) + \Theta(n^2)$.

Le théorème suivant permet de calculer la complexité de la plupart des algorithmes récursifs.

Théorème 1 *Soit $T(n) = aT(n/b) + f(n)$ avec $T(1) = O(1)$ et $f(1) = 0$ et où $a, b, f(n)$ sont des entiers et n est une puissance de b, alors :*

(1) $T(n) = \Theta(n^{\log_b a})$ si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une constante $\varepsilon > 0$;

(2) $T(n) = \Theta(n^{\log_b a} \lg n)$ si $f(n) = \Theta(n^{\log_b a})$;

(3) $T(n) = \Theta(f(n))$ si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une constante $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et pour n assez grand.

Preuve. $T(n) = \Theta(n^{\log_b a}) + g(n)$, avec $g(n) := \sum_{j=0}^{j=\log_b n-1} a^j f(n/b^j)$, puisque :

$$\begin{aligned} T(n) &= aT(n/b) + f(n) = a\big(aT((n/b)/b) + f(n/b)\big) + f(n) \\ &= a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\ &= \vdots \\ &= a^{\log_b n}T(1) + a^{\log_b n-1}f(n/a^{\log_b n-1}) + a^{\log_b n-2}f(n/a^{\log_b n-2}) + \ldots + af(n/b) + f(n) \end{aligned}$$

et

$$a^{\log_b n} = a^{\log_b a \times \log_a n} = (a^{\log_a n})^{\log_b a} = n^{\log_b a}.$$

Dans le cas (1), on a

$$\begin{aligned} g(n) &= \sum_{j=0}^{j=\log_b n-1} a^j O\Big(\big(n/b^j\big)^{\log_b a-\varepsilon}\Big) \\ &= O\Big(\sum_{j=0}^{j=\log_b n-1} a^j \big(n/b^j\big)^{\log_b a-\varepsilon}\Big). \end{aligned}$$

D'où $T(n) = \Theta(n^{\log_b a})$ car $g(n) = O(n^{\log_b a})$ d'après le calcul suivant :

$$\begin{aligned} \sum_{j=0}^{\log_b n-1} a^j \Big(\frac{n}{b^j}\Big)^{\log_b a-\varepsilon} &= n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \Big(\frac{ab^\varepsilon}{b^{\log_b a}}\Big)^j \\ &= n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} (b^\varepsilon)^j \\ &= n^{\log_b a-\varepsilon} \Big(\frac{b^{\varepsilon \log_b n}-1}{b^\varepsilon-1}\Big) \\ &= n^{\log_b a-\varepsilon} \Big(\frac{n^\varepsilon-1}{b^\varepsilon-1}\Big) \\ &= n^{\log_b a-\varepsilon} O(n^\varepsilon) \end{aligned}$$

Dans le cas (2) on a $T(n) = \Theta(n^{\log_b a} \lg n)$ car $g(n) = \Theta(n^{\log_b a} \lg n)$ d'après :

$$\begin{aligned} \sum_{j=0}^{\log_b n-1} a^j \Big(\frac{n}{b^j}\Big)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \Big(\frac{a}{b^{\log_b a}}\Big)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1^j \\ &= n^{\log_b a} \log_b n \end{aligned}$$

Pour le cas (3) il est clair que $g(n) = \Omega(f(n))$, par ailleurs, si $af(n/b) \leq cf(n)$ pour $c < 1$, on a $g(n) = O(f(n))$ d'après :

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\leq f(n) \sum_{j=0}^{\infty} c^j \end{aligned}$$

On a $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$ car $f(n) = \Omega(n^{\log_b a + \varepsilon})$. ∎

Corollaire 1 *Le théorème reste vrai pour tout entier n en supposant seulement que $T(n)$ est une fonction croissante et qu'il existe une constante k telle que $f(bn) \leq kf(n)$ pour n grand.*

Preuve. Si $n_1 = b^k \leq n < b^{k+1} = n_2$ pour un $k \in \mathbb{N}$, alors $n_1 \geq n/b$ et $n_2 \leq nb$.
D'où, dans le cas (1)

$$\Omega(n^{\log_b a}) \leq \Theta(n_1^{\log_b a}) = T(n_1) \leq T(n) \leq T(n_2) = \Theta(n_2^{\log_b a}) \leq O(n^{\log_b a}),$$

dans le cas (2)

$$\Omega(n^{\log_b a} \lg n) \leq \Theta(n_1^{\log_b a} \lg n_1) = T(n_1) \leq T(n) \leq T(n_2) = \Theta(n_2^{\log_b a} \lg n_2) \leq O(n^{\log_b a} \lg n),$$

et dans le cas (3)

$$\Omega(f(n)) = T(n) \leq T(n_2) = \Theta(f(n_2)) \leq O(f(bn)) = O(f(n)).$$

∎

2.2.3 Multiplication rapide

Entiers

Clairement, l'addition et la soustraction de deux nombres de n-digits sont en $\Omega(n)$, et on a $O(n)$ par les algorithmes élémentaires, donc en $\Theta(n)$. En 1952, Kolmogorov avait conjecturé que l'algorithme classique de multiplication en $O(n^2)$ était optimal (qu'on ne pouvait multiplier en moins de $\Omega(n^2)$). En fait, en 1960, Karatsuba a donné un algorithme en $O(n^{\lg 3})$ soit $O(n^{1.58})$. Son fonctionnement est le suivant : Soient N et M deux entiers n-digits à multiplier. Si n est pair, on a $N = a \times 10^{n/2} + b$ et $M = c \times 10^{n/2} + d$ avec a, b, c, d des nombres $\frac{n}{2}$-digits. Si l'on connaissait les trois produits ac, bd et $(a+b)(c+d)$ on pourrait alors obtenir le résultat NM en $O(n)$ car :

$$\begin{aligned} NM &= (a \times 10^{n/2} + b)(c \times 10^{n/2} + d) \\ &= ac \times 10^n + (ad + bc) \times 10^{n/2} + bd \\ &= ac \times 10^n + ((a+b)(c+d) - ac - bd) \times 10^{n/2} + bd. \end{aligned}$$

Si on suppose que n est une puissance de 2, il suffit alors de calculer récursivement les trois produits.

Exemple 1 *Multiplions 1562 et 8943. On a* $1562 = 15 \times 10^2 + 62$ *et* $8943 = 89 \times 10^2 + 43$. *En $O(n)$ on calcul* $15 + 62 = 77$ *et* $89 + 43 = 132$. *Puis avec trois appels récursifs on obtient* $15 \times 89 = 1335$, $62 \times 43 = 2666$, *et* $77 \times 132 = 10164$. *On combine les trois produits pour obtenir le résultat final* $1562 \times 8943 = 1335 \times 10^4 + (10164 - 1335 - 2666) \times 10^2 + 2666 = 13968966$

Toutefois dans l'exemple suivant 132 a plus de 2 digits et ainsi le temps d'exécution ne satisfait pas $T(n) = 3T(n/2) + \Theta(n)$, qui prouverait la complexité annoncée. La décomposition suivante satisfait bien l'équation de récurrence :

$$NM = ac \times 10^n - ((a-b)(c-d) - ac - bd) \times 10^{n/2} + bd.$$

Notons que l'algorithme peut être appliqué à la multiplication des polynômes.

Matrices

Le calcul classique de deux matrices $A \in \mathbb{R}^{n \times p}$ et $B \in \mathbb{R}^{p \times m}$ s'effectue en $O(nmp)$ soit en $O(n^3)$ pour des matrices carrées. Comparons les temps de calcul de l'algorithme $O(n^3)$ avec deux implémentations de l'algorithme de Strassen (uniquement pour la multiplication des matrices carrées) :

```
n = 300
classic           took     140ms
Strassen(dynamic) took    5360ms
Strassen(cached)  took     359ms
n = 500
classic           took     672ms
Strassen(dynamic) took   36875ms
Strassen(cached)  took    2219ms
n = 1000
classic           took    6735ms
Strassen(dynamic) took  256093ms
Strassen(cached)  took   15453ms
n = 1300
classic           took   19609ms
Strassen(dynamic) took  262500ms
Strassen(cached)  took   17172ms
```

Le cas (1) du théorème permet d'établir que l'algorithme de Strassen calcul en $O(n^{2.81})$ le produit $C = AB$ de deux matrices carrées A et B, en pratique, ce n'est qu'à partir de grandes tailles qu'il prend son comportement asymptotique. Expliquons son fonctionnement en supposant que les matrices sont de taille 2^k, on a ainsi :

En calculant récursivement les 7 produits suivants de matrices carrées $n/2 \times n/2$

$$
\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})(B_{11}) \\
P_3 &= (A_{11})(B_{12} - B_{22}) \\
P_4 &= (A_{22})(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12})(B_{22}) \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

on obtient le produit initial de matrices carrées $n \times n$ avec les 8 additions/soustractions suivantes :

$$
\begin{aligned}
C_{12} &= P_5 + P_3 \\
C_{21} &= P_2 + P_4 \\
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}
$$

La validité découle de la justesse des ces quatre égalités que l'on pourra vérifier avec la représentation suivante :

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & + & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & + & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & - \\ \cdot & + & \cdot & - \end{pmatrix}$$

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} - & \cdot & + & \cdot \\ - & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

D'où $T(n) = 7T(n/2) + \Theta(n^2)$, et la complexité est $\Theta(n^{\lg 7})$. La mauvaise implémentation crée récursivement les sous-matrices nécessaires tandis que la bonne ne crée qu'une matrice C.

2.3 Arbres

2.3.1 Définition et propriétés

Le concept d'arbre est fondamental pour l'analyse, et en fait aussi et peut-être surtout pour la conception, aussi nous en donnons une définition formelle (d'autres définitions équivalentes seront vues en exercice). Pour cela, nous notons $f^0(x) = x$ puis $f^k(x) = f(f^{k-1}(x))$ pour tout entier $k \geq 1$ et toute fonction $f : E \to F$; remarquons que $f^k(x)$ n'est défini que si $f^{k-1}(x) \in E$.

Définition 3 *L'ensemble vide $\emptyset$ est un arbre, appelé l'*arbre vide. *Soient un ensemble fini T non vide, un élément $r \in T$ et une fonction $p : T \setminus \{r\} \to T$.*

Si pour tout $x \in T \setminus \{r\}$, il existe un entier $k \geq 1$ tel que $p^k(x) = r$, alors T (muni de la fonction p) est un arbre.

On dit alors que r est la *racine de T*, que $p(x)$ est le *père de x*, et que x est un *fils* de $p(x)$. Plus généralement pour $k \geq 1$, $p^k(x)$ est un *ancêtre de x*, et x est un *descendant* de $p^k(x)$. Un élément sans descendant est une *feuille* de l'arbre, sinon c'est un *sommet interne*. (On appelle tous les éléments des sommets.)

Exemple 2 *Un exemple d'arbre est le* tas H_n à n sommets *défini comme l'ensemble $H_n = \{1, 2, \ldots, n\}$ de racine $r = 1$ muni de la fonction père $p(i) = \lfloor \frac{i}{2} \rfloor$ pour tout $i = 2, \ldots, n$.*

Soit $x \in T$, l'entier k tel que $p^k(x) = r$ est appelé la *profondeur du sommet $x \in T$* ; le *chemin de x vers la racine* est alors la suite $x, p(x), p^2(x), \ldots, p^k(x)$. La racine r est le seul sommet de profondeur nulle. La *hauteur $h(x)$ de $x \in T$* est 0 si x est une feuille, sinon c'est la valeur maximum de k tel que $x = p^k(y)$ pour un descendant (feuille) y de x. La *hauteur $h(T)$ (ou juste h) de T* est -1 si $T = \emptyset$, sinon c'est la hauteur de sa racine r.

Un *chemin de x vers une feuille* est une suite $x = v_0, v_1, v_2, \ldots, v_l = y$ de sommets de T telle que y est une feuille et $v_{i-1} = p(v_i)$. La *longueur d'un chemin* est l, ainsi la profondeur de x est la longueur de l'unique chemin de x vers la racine, et la hauteur de x est la longueur maximum d'un chemin de x vers une feuille. Il est alors clair que ni la profondeur ni la hauteur d'un sommet $x \in T$ ne peuvent dépasser la hauteur h de T.

L'ensemble des ancêtres de $x \in T$ est l'ensembles des sommets de $T \setminus \{x\}$ étant sur l'unique chemin de x vers la racine. L'ensemble des descendants de $x \in T$ est l'ensembles des sommets de $T \setminus \{x\}$ étant sur un chemin de x vers une feuille.

Trivialement, T muni de $p : T \setminus \{r\} \to T$ est un arbre si et seulement si il existe un chemin de x vers la racine, pour tout $x \in T \setminus \{r\}$.

Définition 4 *Deux arbres (T_1, p_1) et (T_2, p_2) sont* isomorphes *s'il existe une bijection $\phi : T_1 \leftrightarrow T_2$ telle que $p_1(y) = x$ si et seulement si $p_2(\phi(y)) = \phi(x)$ pour tout $x, y \in T_1$.*

Définition 5 *Soit T un arbre et $x \in T$. Le* sous-arbre de T enraciné en x *est l'arbre de racine x sur l'ensemble T_x des descendants de x incluant x lui-même (muni de la restriction de la fonction $p : T_x \setminus \{x\} \to T_x$).*

On a les définitions équivalentes suivantes d'un arbre.

Proposition 1 *Soit T un ensemble (fini) et une fonction $p : T \setminus \{r\} \to T$. Alors (T, p) est un arbre si et seulement si, pour tout $x \in T$, $p^k(x) = x$ implique $k = 0$.*

Preuve. $\Rightarrow$: Montrons qu'il est contradictoire que T soit un arbre et qu'il existe un entier $k > 0$ tel que $p^k(x) = x$ pour un $x \in T$. On a $x, p(x), p^2(x), \ldots, p^{k-1}(x) \in T \setminus \{r\}$. Puisque $p^k(x) = x$, alors $p^i(x) \neq r$ pour tout entier i. Donc (T, p) n'est pas un arbre.
$\Leftarrow$: Montrons que si T n'est pas un arbre, alors il existe un entier $k > 0$ tel que $p^k(y) = y$ pour un $y \in T$. Il existe un $x \in T \setminus \{r\}$ tel que $p^k(x) \in T \setminus \{r\}$ pour tout $k \in I = \{0, 1, \ldots, |T| - 1\}$. Puisque $|I| > |T \setminus \{r\}|$, par le principe du pigeonhole, il existe $i, j \in I$, $i < j$, tels que $p^i(x) = p^j(x)$. Donc $p^k(y) = y$ avec $y = p^i(x)$ et $k = j - i$. □

Proposition 2 *(Définition récursive d'un arbre)*
Soit un ensemble fini T. Si $|T| \leq 1$, alors T est un arbre. Un arbre sur un ensemble T de cardinalité ≥ 2 est un un couple $(r, \mathcal{P}(r))$ où $\mathcal{P}(r)$ est une partition en arbres non-vide de $T \setminus \{r\}$.

Preuve. $\Rightarrow$: Si (T, p) est un arbre ayant au-moins deux sommets, alors $\mathcal{P}(r) := \{T_{x_1}, \ldots, T_{x_k}\}$ est une partition en arbres non-vides de $T \setminus \{r\}$, où r est la racine de T et $x_1, \ldots, x_k$ ses fils.
$\Leftarrow$: Il suffit de montrer que la procédure récursive suivante construit un arbre (T, p) :

Algorithm 7 CONSTRUCTION-ARBRE$((r, \mathcal{P}(r)))$

if $|T| \geq 2$ **then**
  Soient $r \in T$ et $\mathcal{P}(r) = \{T_{x_1}, \ldots, T_{x_k}\}$ la partition de $T \setminus \{r\}$ telle que T_{x_i} contient $x_i \in T$.
  for $i \leftarrow 1$ à k **do**
    $p(x_i) := r$
    CONSTRUCTION-ARBRE$(x_i, \mathcal{P}(x_i))$

Montrons par induction sur $|T|$ que la fonction p définie par CONSTRUCTION-ARBRE$(r, \mathcal{P}(r))$ est telle que (T, p) est un arbre de racine r. C'est trivialement vrai si $|T| \leq 1$ car l'ensemble de départ de p est vide. Pour $|T| \geq 2$, par induction la fonction p définie par CONSTRUCTION-ARBRE$(x_i, \mathcal{P}(x_i))$ est telle que $(T \setminus \{r\}, p)$ est un arbre de racine x_i, donc la fonction p définit un chemin de x vers x_i pour tout $x \in T_{x_i} \setminus \{x_i\}$, pour $i = 1, \ldots, k$. Donc après l'instruction $p(x_i) := r$, ce chemin va vers la racine r et (T, p) est un arbre. □

Proposition 3 *Soient n entiers $d_1, d_2, \ldots, d_n$. Il existe un arbre $T = \{v_1, \ldots, v_n\}$ tel que v_i a d_i fils si et seulement si $\sum_{i=1}^{n} d_i = n - 1$.*

Preuve. $\Rightarrow$: Par induction sur n. Pour $n = 1$, $r = v_1$ n'a pas de fils, donc $\sum_{i=1}^{n} d_i = d_1 = 0 = n-1$. Pour $n \geq 2$, on peut supposer que v_n est une feuille de père v_1. Soit T' l'arbre obtenu à partir de T en supprimant v_n, où v_i a d'_i fils dans T' pour $i = 1, \ldots, n-1$. Alors, par induction, $\sum_{i=1}^{n-1} d'_i = n-2$. Dans T, v_1 a $d_1 = d'_1 + 1$ fils, v_i a $d_i = d'_i$ fils pour $i = 2, \ldots, n-1$, et v_n a $d_n = 0$ fils. On a bien $\sum_{i=1}^{n} d_i = n - 1$.
$\Leftarrow$: Par induction sur n. Pour $n = 1$, $d_1 = 0$ est bien le nombre de fils de l'arbre T composé de l'unique sommet v_1. Pour $n \geq 2$, on peut supposer que $d_1 \geq 1$ et $d_n = 0$. Soit $d'_1 = d_1 - 1$ et $d'_i = d_i$ pour $i = 1, \ldots, n-1$. Alors, $\sum_{i=1}^{n-1} d'_i = n - 2$, et donc par induction, il existe un arbre $T' = \{v_1, \ldots, v_{n-1}\}$ tel que v_i a d'_i fils. En ajoutant le sommet v_n de père v_1 à T' on obtient l'arbre T recherché. □

Définition 6 *Soit T un arbre. Si tout élément $x \in T$ a au-plus deux fils, un* fils gauche $g(x)$ *et un* fils droit $d(x)$*, alors T est un* arbre binaire *(on note $g(x)$ ou $d(x) = NIL$ si le fils gauche ou droit n'existe pas, respectivement).*

Un arbre T dont les éléments on au-plus d fils est appelé *arbre d-aire*. Par récurrence sur k il est facile de voir qu'un arbre d-aire a au-plus d^k sommets de profondeur k. Donc,

$$|T| \leq \sum_{k=0}^{k=h} d^k = \frac{d^{h+1} - 1}{d - 1} \quad \text{pour tout arbre } d\text{-aire } T \text{ de hauteur } h,$$

Remarquons que l'on a l'égalité si il y a exactement d^k sommets de profondeur k, un tel arbre d-aire est un *tas complet*. Si H_n est un tas complet de hauteur h, alors $n = 2^{h+1} - 1$ d'où l'observation suivante *fondamentale en algorithmique :*

$$h = \lfloor \lg n \rfloor \quad \text{pour tout tas } H_n \text{ de hauteur } h.$$

(On note $\lceil x \rceil$ l'entier supérieur le plus proche de x si x n'est pas entier, sinon $\lceil x \rceil$ vaut x ; et en fait on utilisera les deux notations : $\lceil x \rceil = \min\{n \in \mathbb{N} : x \leq n\}$ et $\lfloor x \rfloor = \max\{n \in \mathbb{N} : n \leq x\}$.)

Elle permet tout à la fois de concevoir des algorithmes récursifs performants et des structures de données performantes, du point de vue de la complexité. Plus généralement, les structures d'arbres binaires telles que la hauteur h de T satisfait

$$h \leq c_1 \lg |T| + c_2 = O(\lg n), \quad \text{pour deux constantes } c_1 \text{ et } c_2,$$

permettent d'élaborer des algorithmes performants.

2.3.2 Tris comparatifs

Algorithm 2 est un algorithme de tri qui n'effectue aucune comparaison entre les éléments d'entrée. Ceux dont, au contraire, le tri repose sur des comparaisons successives de deux éléments de A sont des algorithmes de *tris comparatifs*. On peut montrer que tous ces algorithmes ont une complexité d'au-moins $\Omega(n \lg n)$. Pour cela représentons un tri comparatif par un *arbre de décision* (cf. définition 6).

L'arbre de décision d'un tri comparatif sur un tableau $A = A[1], \ldots, A[n]$ est l'arbre dont les sommets correspondent à tous les états possibles de l'algorithme, chaque état étant défini par l'ensemble des comparaisons déjà effectuées et par la comparaison que l'algorithme effectue alors dans ce cas. La racine correspond à l'état initial. Le fils gauche (resp. droit) d'un sommet correspond à l'état découlant du résultat $<$ (resp. $\geq$) de la dernière comparaison. Les feuilles sont les états finaux, correspondant à l'état où l'on connait le tableau A trié, c'est-à-dire où l'on connait une bijection $\pi : A \leftrightarrow \{1, \ldots, n\}$ telle que $\pi(a) \leq \pi(b)$ si et seulement si $a \leq b$. Il y autant de telles bijections que de permutations de $\{1, \ldots, n\}$. Il existe donc toujours un tableau A tel que le tri comparatif de A effectuera autant de comparaisons que la hauteur h de son arbre de décision.

Puisqu'il y a $n!$ permutations et qu'un arbre binaire de hauteur h a au-plus 2^h feuilles, on a $n! \leq 2^h$ et le résultat découle du fait que

$$h \geq \lg(n!) = \Theta(n \lg n).$$

Clairement $\lg(n!) \leq \lg(n^n) = O(n \lg n)$, la difficultée est de montrer $\lg(n!) = \Omega(n \lg n)$. On pourrait le faire avec une approche de l'analyse classique [1] mais l'approche suivante fonctionne aussi :

$$\begin{aligned}(2k)! &= 2k \times \ldots \times 2k-i+1 \times \ldots \times k+1 \times k \times \ldots \times i \times \ldots \times 1 \\ &= (2k \times 1) \ldots ((2k-i+1) \times i) \ldots ((k+1) \times k) \\ &\geq k^k\end{aligned}$$

En effet $(2k-i+1) \times i = ki + (ki - i^2) + i \geq k$ pour $i = 1, \ldots, k$. D'où $n! \geq (\frac{n}{2})^{\frac{n}{2}}$ donc $\lg(n!) \geq \frac{n}{2}(\lg n - \lg 2) = \Omega(n \lg n)$.

1. Pour cela on prouve l'inégalité préliminaire

$$n! \geq \left(\frac{n}{\mathrm{e}}\right)^n$$

en utilisant le théorème fondamental de l'analyse (cf cours d'analyse) qui permet le calcul intégral. On définit le *logarithme naturel ou de Neper* comme

$$\ln(x) := \int_1^x \frac{1}{t} dt$$

Ainsi $\ln(x)$ vérifie les propriétés des fonctions logarithmes : $\ln(x) = 0$ implique $x = 1$ donc la propriété (i) des logarithmes est vérifiée. La propriété (ii) est aussi vérifiée car : (on suppose $x, y > 1$)

$$\begin{aligned}\ln(xy) := \int_1^{xy} \frac{1}{t} dt &= \int_1^x \frac{1}{t} dt + \int_x^{xy} \frac{1}{t} dt \\ &= \int_1^x \frac{1}{t} dt + \int_1^y \frac{1}{vx} x dv \\ &= \int_1^x \frac{1}{t} dt + \int_1^y \frac{1}{v} dv =: \ln(x) + \ln(y)\end{aligned}$$

(Avec $v := t/x$ on a $dt/dv = x$). La propriété (iii) se montre de manière similaire :

$$\begin{aligned}\ln(x/y) := \int_1^{x/y} \frac{1}{t} dt &= \int_1^x \frac{1}{t} dt - \int_{x/y}^x \frac{1}{t} dt \\ &= \int_1^x \frac{1}{t} dt - \int_1^y \frac{1}{ux/y} x/y \, du \\ &= \int_1^x \frac{1}{t} dt - \int_1^y \frac{1}{u} du =: \ln(x) - \ln(y)\end{aligned}$$

(Avec $u := ty/x$ on a $dt/du = x/y$). Donc il existe une constante, notée $\mathrm{e} := \ln^{-1}(1)$ (constante de Neper), telle que $\ln(x) = y$ tel que $\mathrm{e}^y = x$. On a ensuite

$$\begin{aligned}\ln n! &= \sum_{x=2}^{x=n} \ln x \\ &\geq \int_1^n \ln x \, \mathrm{d}x \\ &= [x \ln x - x]_1^n \\ &= n \ln n - n + 1 \\ n! &\geq \mathrm{e}^{n \ln n - n + 1} \\ &= \mathrm{e}\left(\frac{n}{\mathrm{e}}\right)^n,\end{aligned}$$

et donc

$$\begin{aligned}h &\geq \lg(n!) \\ &\geq \lg\left(\frac{n}{\mathrm{e}}\right)^n \\ &= n \lg n - n \lg \mathrm{e} \\ &= \Omega(n \lg n).\end{aligned}$$

2.4 Algorithmique théorique

2.4.1 Problème de l'arrêt

Version algorithmique de l'argument diagonal du théorème de Cantor. Autre argument : Si il existe une bijection $\phi : \mathbf{N} \leftrightarrow \mathcal{P}(\mathbf{N})$ alors on a une contradiction avec "$a \in X$?" pour $X = \{n \in \mathbf{N} : n \notin \phi(n)\}$ et $a = \phi^{-1}(X)$. Notez aussi que $f(x, y) = 2^y(2x + 1) - 1$ est une bijection $\mathbf{N} \times \mathbf{N} \leftrightarrow \mathbf{N}$.

Montrons qu'il ne peut pas exister un algorithme certifiant que l'exécution d'un programme donné avec une entrée donnée s'arrête ou non.

Un programme P et une entrée I pour ce programme peuvent être codés par un même type, disons `code`. Suposons qu'il existe une procédure `HALT(code P, code I)` qui prenne en entrée un programme P et une entrée I et qui retourne **true** si le programme P s'arrête avec l'entrée I, et **faux** sinon. Alors on peut construire la procédure suivante :

```
int TROUBLE(code d) {
  if (HALT(d,d))
    return 1;
  else
    while(true);
}
```

Soit `code t` le code de `TROUBLE(code d)`, on obtient une contradiction à l'exécution de `TROUBLE(t)`. En effet, si l'exécution s'arrête le test `if` est faux et l'exécution ne s'arrête pas. Cette exécution ne peut donc pas s'arrêter mais alors le test `if` est vrai et elle s'arrête.

2.4.2 Combinatoire

Combinatoire et géométrie

(0) Soient cinq points dans un triangle équilatéral de côté 1. Montrer qu'il en existe deux à distance $\leq 1/2$.

(1) *Soient un ensemble X de n points rouges et un ensemble Y de n points verts dans le plan tels que les $2n$ points soient distincts. Existe-t-il une bijection $\phi : X \leftrightarrow Y$ entre les points rouges et les verts telle qu'aucune paire de segments ne se croisent parmis les n segments $[x, y = \phi(x)]_{x \in X}$?*

(2) *Soit E un ensemble fini de points du plan euclidien contenant au-moins trois points non-alignés. Peut-on toujours trouver une droite du plan contenant* exactement *deux points dans E ?*

(3) *Soient trois ensembles disjoints A_1, A_2, A_3 de même cardinalité n et une relation symétrique $n + 1$-régulière, c'est-à-dire qui relie chaque élément de A_i à $n + 1$ éléments de A_j pour $j \in \{1, 2, 3\} \setminus \{i\}$, et pour $i = 1, 2, 3$. Peut-on toujours trouver trois éléments $x_i \in A_i$ deux-à-deux reliés (pour $i = 1, 2, 3$) ?*

1) Oui. La solution à cette question passe par l'introduction de la valeur $L_\phi = \sum_{x \in X} ||x - \phi(x)||$ de la longueur totale des n segments.) Il suffit ensuite de voir que la bijection ϕ telle que L_ϕ est minimum est une solution, en effet les inégalités triangulaires impliquent que si il y avait deux segments $[x_1, y_1]$ et $[x_2, y_2]$ se croisant alors la bijection ϕ' telle que $\phi'(x_1) = y_2$, $\phi'(x_2) = y_1$ et $\phi'(x) = \phi(x)$ pour $x \in X \setminus \{x_1, x_2\}$ serait telle que $L_{\phi'} < L_\phi$.

2) Oui. Soit (d, x) tel que d est une droite contenant au-moins deux points de E et tel que $x \in E \setminus d$ minimise sa distance à d. Alors d contient exactement deux points de E car si d contenait (au-moins) trois points de P on aurait une contradiction avec le fait que la distance de x à d est minimum. En effet si on a $r, s, t \in E \cap d$ avec $s \in]r, t[$, alors soit y le point de d le plus proche de x, on peut supposer sans perte de généralité que $s \in]r, y[$. Mais alors s est plus proche de la droite (rx) que x n'est proche de d.

3) Oui. Soit $x_i \in A_i$ un élément qui est relié avec le minimum δ d'éléments de A_j pour $i, j \in \{1, 2, 3\}$ et $i \neq j$. Puisque x_i a $n+1$ relations, on a $\delta \geq 1$, et donc x_i est relié à un élément $x_j \in A_j$.

De plus, x_i est relié à $n-\delta+1$ éléments de A_k avec $k \in \{1,2,3\}\setminus\{i,j\}$. Mais x_j est relié à au-moins δ éléments de A_k donc il existe un $x_k \in A_k$ relié à x_j et à x_i.

Invariant combinatoire

Nous avons vu déjà quelques preuves de validité utilisant la notion d'invariant, illustrons la puissance de cette notion pour appréhender des questions d'ordre algorithmique. Soit la classe Java suivante :

```
public class LifeMatrix {
  public static void main(String[] args) {
    final int n = Integer.parseInt(args[0]);
    final int x = Integer.parseInt(args[1]);
    final int seed = Integer.parseInt(args[2]);
    boolean[][] mat = new boolean [n][n];
    final Random generator = new Random(seed);
    for(int i = 0 ; i < x ; i++)
      mat[generator.nextInt(maxN)][generator.nextInt(maxN)]=true;
    boolean test=true;
    while(test){
      test=false;
        for(int i = 0 ; i < n ; i++){
          for(int j = 0 ; j < n ; j++){
            if(!mat[i][j]){
              int vois=0;
            if(i>0 && mat[i-1][j])
              vois++;
            if(i+1<n && mat[i+1][j])
              vois++;
            if(j>0 && mat[i][j-1])
              vois++;
            if(j+1<n && mat[i][j+1])
              vois++;
            if(vois >= 2){
              mat[i][j]=true;
              test=true;
            }
          }
        }
      }
    }
  }
}
```

Après ajout d'une fonction d'affichage de `mat[][]` avec `X` pour `true` et `0` pour `false`, à chaque passage dans la boucle **while**, après les commandes

```
javac LifeMatrix.java
java LifeMatrix 5 9 1276871
```

on a comme sortie écran :

```
  X 0 0 0 X   X 0 0 0 X   X X X X X   X X X X X
  0 0 0 0 0   X X X 0 X   X X X X X   X X X X X
  X X X 0 X   X X X X X   X X X X X   X X X X X
  0 0 0 X X   0 0 X X X   0 X X X X   X X X X X
  0 0 0 X 0   0 0 X X X   0 X X X X   X X X X X
```

Notons que si `mat[i][i]=true` pour $i = 0, \ldots, n-1$ avant la boucle **while** alors tous les éléments de `mat[][]` sont `true` au sortir de la boucle. Montrons que si `args[1]==args[0]-1`, c'est-à-dire si $x = n - 1$, alors, à la fin de l'exécution, il existe au-moins un élément `false` dans `mat[][]`.

Soit A^i la grille de $n \times n$ cases carrées `X` ou `0` affichée après le $i$ème passage dans la boucle **while**. On prouve l'invariant suivant : *soit $\pi(A^i)$ le périmètre de la surface des cases `X`, alors, à chaque instant de l'exécution de l'algorithme, on a $\pi(A^i) \leq \pi(A^0)$.* Lorsqu'une case ij passe de `0` à `X` c'est qu'elle a au-moins deux cases voisines `X`. Donc le périmètre ne peut augmenter : $\pi(A^{i+1}) \leq \pi(A^i)$; l'invariant est vrai. Avec l'instruction d'origine `int a = n-1`, on a $\pi(A^0) \leq 4(n-1)$ puisque que le périmètre est inférieur à 4 fois le nombre de cases `X`. Quand tous les éléments de `mat[][]` sont `true`, on a un périmètre de $4n$, ce qui implique $\pi(A^i) = 4n$ pour un certain i assez grand, or c'est impossible par l'invariant ; fin de la preuve.

Chapitre 3

Outils de conception algorithmique

Un algorithme est bien conçu si aucun autre algorithme ne peut faire strictement la même chose avec une meilleure complexité ou en utilisant moins d'espace mémoire. En particulier, les structures de données permettent d'effectuer finement des opérations récurrentes telles que la recherche d'un élément maximum ou minimum.

3.1 Récursivité

Puissance

```
static double PowIt (double x, int n){
  double res = x;
  while(n-- > 1)
    res *= x;
  return res;
}
```

La fonction `PowIt` calcule en $\Theta(n)$ la puissance x^n, mais on pourrait aussi procéder récursivement.

```
static double PowRec (double x, int p){
  if (p == 1) return x;
  else
    return PowRec(x, p/2 ) * PowRec(x, p/2 + (p%2) );
}
```

Le temps d'exécution $T(p)$ de `PowRec` satisfait l'équation de récurrence $T(p) = 2T(p/2) + \Theta(1)$.

Le cas (1) du théorème implique que `PowRec` a la même complexité $\Theta(n)$ que `PowerIt`. En pratique la récursivité fait perdre du temps si elle n'améliore pas la complexité, mais avec le calcul récursif suivant :

```
static double PowRecRap (double x, int p){
  if (p == 1) return x;
  else{
    if(p%2 == 0)
      return PowRecRap(x*x,p/2);
    else
      return x * PowRecRap(x*x,p/2);
  }
}
```

On a les temps suivants pour le calcul de 2 puissance 2.10^9 :

```
2.0 power 2000000000 Rec.        took 51546ms
2.0 power 2000000000 It.         took  6781ms
2.0 power 2000000000 Rec. Rap.   took     0ms
```

Le temps $T(p)$ d'exécution de `PowRecRap (double x, int p)` satisfait $T(p) = T(p/2) + \Theta(1)$ (en supposant la multiplication est constante) ce qui donne une complexité de $\Theta(\lg p)$ par le cas (2) du théorème. En fait si on inclut le temps de la multiplication avec x un nombre n-digit on obtient $O(n^{1.58} \lg p)$. Notez que, en calculant les k premières puissances on obtient une procédure vérifiant $T(n) = T(n/k) + \Theta(1)$, ce qui peut être plus rapide en pratique que `PowRecRap` (pour laquelle $k = 2$) mais c'est équivalent au niveau de la complexité. Remarquons aussi que, avec une approximation pour ϕ, on peut calculer Fibonacci F_n en $\Theta(\log n)$ avec **Algorithm 5**. Pour un résultat précis, on peut utiliser l'égalité suivante :

$$\text{Avec } X := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{on a} \quad \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = X^n \quad \forall n \geq 1$$

Remarquons que ϕ et $\bar{\phi}$ sont les valeurs propres de X. Donc si on diagonalise X, le calcul de X^n revient au calcul ϕ^n et $\bar{\phi}^n$ qui est compliqué par l'irrationalité de $\sqrt{5}$. En adaptant `PowRecRap` pour les matrices nous calculons X^n en effectuant $\lg n$ multiplications de matrices 2×2. On obtient donc une complexité de $\Theta(\lg n)$ si on considère la multiplication d'entiers comme constante. Toutefois les composantes de X^n sont grandes puisque $F_n = \Theta(\phi^n)$ donc F_n est $O(n)$-digits. Si l'on tient compte du temps de multiplications des entiers on a une complexité de $O(n^{1.58} \lg n)$. En fait, à chaque étape i de la récursion on effectue $8 = O(1)$ produits de $\frac{n}{2^i}$-digits, donc un calcul plus précis donne

$$\sum_{i=1}^{i=n} \left(\frac{n}{2^i}\right)^{1.58} = n^{1.58} \times \sum_{i=1}^{i=n} \left(\frac{1}{2^{1.58}}\right)^i = O(n^{1.58})$$

3.2 Backtracking

Sous ce terme on désigne les méthodes parcourant un arbre d'exploration. Un exemple est la recherche des placements possibles de 8 reines sur un échiquier sans qu'elles ne se menacent (avec la règle du jeu d'échec).

```
public class EightQueens {
  static int SIZE;              // La taille de l'échiquier est de SIZE*SIZE cases
  int [] solution;
  boolean [] ligne, diag1, diag2;

  public EightQueens(int S){
    SIZE=S;
    solution = new int[SIZE]; // La reine en colonne c est sur la ligne solution[c]
    ligne = new boolean[SIZE];                    //  Test si la ligne l est libre
    diag1 = new boolean[2*SIZE-1];            // Test si la diagonale l+c est libre
    diag2 = new boolean[2*SIZE-1];     // Test si la diagonale l-c+SIZE-1 est libre
    for (int j=0; j < SIZE; ++j)
      ligne[j]=diag1[j]=diag2[j]=true;
    for (int j=SIZE; j < 2*SIZE-1; ++j)
      diag1[j]=diag2[j]=true;
  }
  void essayer(int c){
    for (int l=0; l < SIZE; ++l){      //Recherche une solution avec la reine (l,c)
      if(ligne[l] && diag1[l+c] && diag2[l-c+SIZE-1]){
        solution[c]=l;
        ligne[l]=diag1[l+c]=diag2[l-c+SIZE-1]=false;
        if(c!=SIZE-1) essayer(c+1);               //  Test faux = solution trouvée
```

```
        ligne[l]=diag1[l+c]=diag2[l-c+SIZE-1]=true;        // Efface la reine (l,c)
      }
    }
  }
  public static void main(String[] args){
    long start,finish;
    final int TRY=15;
    for(int i = 8; i < TRY; ++i){
      start= System.currentTimeMillis();
      EightQueens Q = new EightQueens(i);
      Q.essayer(0);
      finish= System.currentTimeMillis();
      System.out.println("chess board " + i + "*" + i + " took " + (finish - start) + "ms");
    }
  }
}
```

Les premières solutions trouvées pour les échiquiers de taille 8x8 à 11x11 sont :

```
0 4 7 5 2 6 1 3
0 2 5 7 1 3 8 6 4
0 2 5 7 9 4 8 1 3 6
0 2 4 6 8 10 1 3 5 7 9
```

La sortie écran de `EightQueens` donne les temps de calculs de toutes les solutions :

```
chess board 8*8   took       0ms
chess board 9*9   took       0ms
chess board 10*10 took      15ms
chess board 11*11 took      31ms
chess board 12*12 took     188ms
chess board 13*13 took    1015ms
chess board 14*14 took    6281ms
chess board 15*15 took   40716ms
chess board 16*16 took  279507ms
chess board 17*17 took 2034192ms
```

Théoriquement, la taille de l'arbre d'exploration serait $n^{n+1} - 1$ s'il était complet, puisque le premier test est faux avec `l==solution[c-1]` on recherche au maximum $n! = \Theta(n^n)$ reines `(l,c)`, donc la complexité est au-plus $O(n^n)$, l'utilisation est rapidement inutilisable en pratique. En ne recherchant qu'une solution on "coupe" beaucoup de branches de l'arbre d'énumération ; on obtient les temps suivants :

```
chess board 8*8   took       0ms
chess board 9*9   took       0ms
chess board 10*10 took       0ms
chess board 11*11 took       0ms
chess board 12*12 took      16ms
chess board 13*13 took      93ms
chess board 14*14 took     360ms
chess board 15*15 took    3078ms
chess board 16*16 took   19297ms
chess board 17*17 took  100359ms
chess board 18*18 took  976078ms
chess board 19*19 took 5779000ms
```

On ne peut éviter "l'explosion combinatoire".

Point 3 *On se demandera toujours si l'algorithme conçu est de complexité exponentielle.*

3.3 Structures de données

3.3.1 Motivation

Soit V un ensemble de sommets et $A \subseteq V \times V$ un ensemble d'arcs (*i.e.* de couples de sommets). Un chemin dans G est une suite $P = (v_0, v_1, \ldots, v_m)$ où $v_0, \ldots, v_m \in V$ sont différents et $v_{i-1}, v_i) \in A$; pour deux sommets $s, t \in V$, le chemin P est un $s-t$ chemin si $s = v_0$ et $t = v_m$. On munit G d'une fonction de longueur $l : A \to \mathbb{Q}_+$. La longueur d'un chemin $P = (v_0, v_1, \ldots, v_m)$ est $l(P) := \sum_{i=1}^{m} l(v_{i-1}, v_i)$. La *distance de s à t* est la longueur minimum d'un $s-t$ chemin.

Soit $s \in V$, l'algorithme 8 de Dijkstra (1959) donne un plus court $s-t$ chemin pour tout $t \in V$.

Algorithm 8 DIJKSTRA(G, l, s)

Initialiser $U := V$, $f(s) := 0$ et $f(v) := \infty$ pour tout $v \in V \setminus \{s\}$.
while $U \neq \emptyset$ **do**
  Trouver $u \in U$ minimisant $f(u)$.
  for all $a = (u, v) \in A$ tel que $f(v) > f(u) + l(a)$ **do**
    Réinitialiser $f(v) := f(u) + l(a)$.
  Réinitialiser $U := U \setminus \{u\}$.

Théorème 2 *Après l'exécution de l'algorithme de Dijkstra, $f(t)$ est la distance de s à t pour tout $t \in V$.*

Preuve. Soit $d(v)$ la distance de s à $v \in V$. Clairement, $d(v) \leq f(v)$ à chaque itération. Il suffit de montrer que

$$d(v) = f(v) \text{ pour tout } v \in V \setminus U \text{ à chaque itération.}$$

C'est vrai au début car $U = V$. En raisonnant par induction, il suffit ainsi de montrer que $d(u) = f(u)$ pour le sommet u trouvé en début de boucle while. Supposons par l'absurde que $d(u) < f(u)$. Soit $s = v_0, v_1, \ldots, v_k = u$ un plus court $s-u$ chemin et soit i le plus petit indice tel que $v_i \in U$. Montrons que alors $d(v_i) = f(v_i)$. Si $i = 0$, c'est vrai car $v_i = s$ et $f(s) = 0$. On a donc $i > 0$. Alors $d(v_i) = d(v_{i-1}) + l(v_{i-1}, v_i)$ car $v_0, v_1, \ldots, v_k$ est un plus court $s-u$ chemin. Puisque $v_{i-1} \in V \setminus U$, par induction, on a $d(v_{i-1}) = f(v_{i-1})$. De plus, v_{i-1} a été le sommet u, et donc $f(v) \leq f(v_{i-1}) + l(v_{i-1}, v)$ pour tout v tel que $(v_{i-1}, v) \in A$. D'où

$$d(v_i) = d(v_{i-1}) + l(v_{i-1}, v_i) = f(v_{i-1}) + l(v_{i-1}, v_i) \geq f(v_i)$$

Mais alors $f(u) > d(u) \geq d(v_i) \geq f(v_i)$, ce qui est impossible d'après la règle du choix de u. ∎

L'algorithme de Dijkstra est en $O(|V|^2)$ puisqu'on passe $|V|$ fois dans la boucle while dans laquelle la recherche de u est en $O(|V|)$, et le temps total de la mise à jour de $f()$ est $O(|A|) = O(|V|^2)$. Toutefois, supposons que l'on puisse construire en $O(|V| \lg |V|)$ une structure de donnée contenant les valeurs de $f()$ retournant le u minimum en $O(1)$ et supportant la mofication de $f(v)$ en $O(\lg |V|)$, on obtient alors une implémentation de Dijkstra en $O(|A| \log |V|)$. Une telle structure est une *file de priorité*.

3.3.2 Tas et file de priorité

Point 4 *L'utilisation de structures de données arborescente de hauteur $O(\lg n)$ peut faire diminuer la complexité de l'algorithme.*

Un tableau A est une suite finie de $n := longueur[A]$ éléments dont la valeur du $i$ème élément est notée $A[i]$. Tout tableau A induit un arbre binaire $T = \{1, \ldots, n\}$ de racine 1 par la fonction *père* $p(i) = \lfloor i/2 \rfloor$ définie pour tout indice i de 2 à n. Si $2i \leq n$, l'élément i a un *fils gauche* $g(i) := 2i$, si $2i + 1 \leq n$, alors i a un *fils droit* $d(i) := 2i + 1$, et si $2i > n$, alors i est une feuille de T.

Définition 7 *Si $A[p(i)] \leq A[i]$ pour tout $i = 1, \ldots, n$, alors A est un* tas *(où $\leq$ est une relation d'ordre total sur A).*

D'après **Proposition 4** ci-dessous, on peut réordonner tout tableau de façon à en faire un tas en appliquant **Algorithm 9**.

Algorithm 9 CONSTRUIRE-TAS(A)

```
taille[A] ← longueur[A]
for i ← ⌊longueur[A]/2⌋ à (décroître) 1 do
    ENTASSER(A, i)
```

Algorithm 10 ENTASSER(A, i)

```
l ← 2i
r ← 2i + 1
if l ≤ taille[A] et A[l] < A[i] then
    min ← l
else
    min ← i
if r ≤ taille[A] et A[r] < A[min] then
    min ← r
if min ≠ i then
    échanger A[i] et A[min]
    ENTASSER(A, min)
```

Propriété 1 *Soit T l'arbre binaire induit par un tableau A et soit $i \in T$. Si $T_{g(i)}$ et $T_{d(i)}$ sont des tas, alors, après l'exécution de ENTASSER(A, i), T_i est un tas.*

Preuve. Si $A[i] \leq A[g(i)]$ et $A[i] \leq A[d(i)]$, alors T est un tas. Sinon, on peut supposer que $A[i] > A[g(i)]$ avec $A[g(i)] \leq A[d(i)]$. Après l'échange de $A[i]$ et $A[g(i)]$, T est un tas. □

Proposition 4 *Après l'exécution de CONSTRUIRE-TAS(A), A est un tas.*

Preuve. Il suffit de montrer que, après chaque passage dans la boucle **for**, T_j est un tas pour $j = i, \ldots, n$ (avec $n := longueur[A]$). Comme $g(j) > n$ pour $j = \lfloor n/2 \rfloor + 1, \ldots, n$, on a initialement $|T_j| = 1$ et donc que T_j est un tas. Par induction, au $i$ème passge dans la boucle, $T_{g(i)}$ et $T_{d(i)}$ sont des tas, et donc le résultat découle de **Propriété 1**. □

Analysons la complexité de l'algorithme de construction d'un tas.

Lemme 1 *Soit T (l'arbre d') un tas de hauteur h_T avec n_T sommets, et soit $n_T(h)$ le nombre de sommets de T à la hauteur h, alors :*

$$n_T(h) \leq \left\lceil \frac{n_T}{2^{h+1}} \right\rceil$$

Preuve. C'est clairement vrai si $n_T \leq 1$ ou si $h = h_T$, supposons donc le contraire. Soient T_g et T_d les deux sous-arbres de T enracinés en $g(1) = 2$ et $d(1) = 3$ respectivement (1 est la racine de

T). Si T_g est complet on a :

$$\begin{aligned} n_T(h) &= n_{T_g}(h) + n_{T_d}(h) \\ &= 2^{h_{T_g}-h} + n_{T_d}(h) \\ &\leq 2^{h_{T_g}-h} + \left\lceil \frac{n_{T_d}}{2^{h+1}} \right\rceil \\ &= \left\lceil 2^{h_{T_g}-h} + \frac{n_{T_d}}{2^{h+1}} \right\rceil = \left\lceil \frac{2^{h_{T_g}+1} + n_{T_d}}{2^{h+1}} \right\rceil \\ &= \left\lceil \frac{n_{T_g} + 1 + n_{T_d}}{2^{h+1}} \right\rceil = \left\lceil \frac{n_T}{2^{h+1}} \right\rceil \end{aligned}$$

Si T_g n'est pas complet, c'est que T_d est complet et alors, de la même manière, on a le résultat. □

Lemme 2 *Pour tout $0 < x < 1$, on a $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$.*

Preuve. En dérivant $\sum_{k=0}^{\infty} x^k = 1/(1-x)$ on a $\sum_{k=0}^{\infty} kx^{k-1} = 1/(1-x)^2$. D'où le résultat en multipliant chaque terme par x. □

Proposition 5 *La complexité de CONSTRUIRE-TAS(A) est $\Theta(n)$.*

Preuve. Clairement la complexité est $\Omega(n)$ de par la boucle **for**. Il est facile de voir (par induction sur h) que la complexité de ENTASSER(A, i) est $O(h)$, où h est la hauteur de i. Le temps total pour CONSTRUIRE-TAS(A) est donc (en utilisant les deux lemmes) :

$$\begin{aligned} O(1) + \sum_{h=0}^{h=h_T} n_T(h)O(h) &= O(\sum_{h=0}^{h=h_T} h \times n_T(h)) \\ &\leq O(n \sum_{h=0}^{\infty} h(1/2)^h)) \\ &= O(2n) = O(n) \end{aligned}$$

□

Algorithm 11 EXTRAIRE-MIN-TAS(A)

```
if taille[A] < 1 then
   error : Underflow
min ← A[1]
A[1] ← A[taille[A]]
taille[A] ← taille[A] − 1
ENTASSER(A, 1)
return min
```

3.3.3 Arbre rouge-noir

Définition 8 *Un arbre binaire T est un* arbre binaire de recherche *si pour tout $x \in T$, $y \in T_{g(x)}$ et $z \in T_{d(x)}$, on a $y \leq x \leq z$ (où $\leq$ est une relation d'ordre totale sur T).*

Définition 9 *Soit T un arbre binaire dont chaque élément est rouge ou noir. Si T satisfait les propriétés suivantes alors c'est un* arbre rouge-noir *:*
1) Si $x \in T$ est rouge, alors aucun de ses fils n'est rouge ;
2) La définition suivante de la fonction hauteur noire *$hn(x)$ sur T est consistante :*

Algorithm 12 MODIFIE-TAS(A, i, key)

if $key > A[i]$ **then**
  $A[i] \leftarrow key$
  ENTASSER(A, i)
if $key < A[i]$ **then**
  while $i > 1$ et $key < A[p(i)]$ **do**
    $A[i] \leftarrow A[p(i)]$
    $i \leftarrow p(i)$
  $A[i] \leftarrow key$

Algorithm 13 INSERTION-TAS(A, key)

$taille[A] \leftarrow taille[A] + 1$
$A[taille[A]] \leftarrow \infty$
MODIFIE-TAS$(A, taille[A], key)$

$$hn(x) = \begin{cases} 1 & \textit{si } x \textit{ n'a pas deux fils,} \\ hn(y) & \textit{si } x \textit{ a un fils } y \textit{ rouge,} \\ hn(y) + 1 & \textit{si } x \textit{ a un fils } y \textit{ noir.} \end{cases}$$

L'intérêt des arbres rouge-noir est qu'il existe des algorithmes en $O(\lg n)$ permettant d'insérer ou de supprimer un élément, ainsi que la recherche d'un élément minimum ou maximum.

Bibliographie

[1] Brun Baynat, Philippe Chrétienne, Claire Hanen, Safia Kedad-Sidhoum, Alix Munier-Kordon et Christophe Picouleau, *Exercices et Problèmes d'Algorithmique*, Dunod (2003).

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein, *Introduction to Algorithms*, MIT Press (2001)

[3] Denis Cornaz, *cours/TD école ingénieur Polytech' Clermont* (2009)

[4] Vincent Granet, *Algorithmique et programmation en Java*, Dunod (2004).

[5] Anatoly A. Karatsuba, Y. Ofman, "Multiplication of multi-digit numbers on automata", *Soviet Physics Doklady 7*, pp. 595-596 (1963).

[6] E. Lionnet, "Note sur la limite du nombre des divisions à faire pour trouver le plus grand commun diviseur de deux nombres entier", *Nouvelles annales de mathématiques 1er série*, tome 4 (1845), p. 617-626.

[7] Bernard Quément, *cours/TD université Paris-Dauphine* (2009).

[8] Ivan Stojmenovic, "Recursive Algorithms in Computer Science Courses : Fibonacci Numbers and Binomial Coefficients", *IEEE transactions on education 43*(3) p. 273-276 (2000).

[9] Volker Strassen, "Gaussian elimination is not optimal", *Numer. Math. 13*, 354-356 (1969).