INTRODUCTION A L'ALGORITHMIQUE

Albin MORELLE

ESIEE – Mars 2010

Version 1.3

SOMMAIRE

1 CONCEPTION ET ÉVALUATION D'UN ALGORITHME

1.1 Algorithme vs programme – Itératif vs récursif

1.1.1 Définitions

Un **algorithme**[1] [2] est une procédure mécanique qui termine en un temps fini d'étapes. C'est le squelette abstrait d'un **programme** informatique, indépendant du mode de codage particulier qui permettra sa mise en œuvre effective au sein d'un ordinateur.

Par la suite, on utilisera indifféremment les termes d'algorithme et de programme.

Les langages de programmation offrent tous des structures de programme itératives (boucles). Un programme qui en emploie est dit **itératif.**

Un sous-programme (procédure ou fonction) est dit **récursif** s'il est susceptible de s'appeler lui-même directement (récursivité directe) ou indirectement via l'appel d'autres sous-programmes (récursivité indirecte).

Tout sous-programme récursif a la structure générale suivante :

```
Procédure P (donnée de taille n)
    Si condition d'arrêt alors résultat       // décision
    Sinon P(donnée de taille inférieure à n)  // appel récursif
FinProcédure
```

De façon générale, l'exécution d'un programme récursif met en œuvre, sous jacente, une pile. De tels programmes s'exécutent donc à espace mémoire non constant. Il existe toutefois un cas particulier de récursivité qui peut s'exécuter sans pile, à espace mémoire constant : la récursivité terminale.

Un appel récursif est dit terminal quand il est la dernière instruction exécutée dans le cas considéré. Un sous-programme récursif est dit **récursif terminal** quand chacun de ses appels récursifs est terminal. Ainsi, par exemple, une fonction récursive terminale *f* est une fonction dans laquelle tout appel récursif est de la forme *return f(...)*, sans combinaison du résultat de l'appel récursif avec d'autres valeurs.

---

[1] Le mot « **algorithme** » vient du nom du mathématicien perse Al Khwarizmi (~783 – ~850), qui est demeuré célèbre pour avoir introduit dans le monde arabo-musulman et en Occident la numération décimale (rapportée des Indes). Il est notamment l'auteur d'un ouvrage de procédés de calculs dont le titre contient l'expression « Al-jabr », qui sera traduite en latin par « algebra » et donnera plus tard le mot « **algèbre** ». Latinisé, le nom d' Al Khwarizmi donnera progressivement naissance au mot Algorithme. Algorithme désignera d'abord le système constitué du zéro, des neuf chiffres et des méthodes de calcul d'origine indienne, avant d'acquérir progressivement l'acceptation plus large et plus abstraite d'aujourd'hui.

[2] « *Un bon algorithme est comme un couteau tranchant : il fait exactement ce que l'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant* »[3]

Exemple

*L'appel récursif de la factorielle : return n * fact(n-1) ;*
n'est pas terminal, puisque la dernière opération exécutée sera la multiplication.

Par contre, la factorielle suivante est une fonction récursive terminale :

```
// @param n≥0, a=1
// @return  n! * a
int fact(int n, int a) {
      if  ( n <= 1 ) return a ;
      else return fact(n-1, n*a) ;
}
```

Le paramètre `a` *y joue le rôle d'accumulateur. Il doit être initialisé à 1, élément neutre de la multiplication, lors du premier appel. Par exemple le calcul de 5! s'obtient par exécution de : fact(5,1).*

Un sous-programme récursif terminal peut facilement être « dérécursivé », c'est-à-dire transformé en programme itératif sans pile. Certains compilateurs savent détecter la récursivité terminale et implémenter automatiquement de tels sous-programmes sous forme itérative afin d'optimiser l'exécution (c'est le cas notamment pour les langages fonctionnels, comme par exemple OCaml), mais tous (et pas des moindres comme le compilateur Java de Sun) ne savent pas le faire[1] !

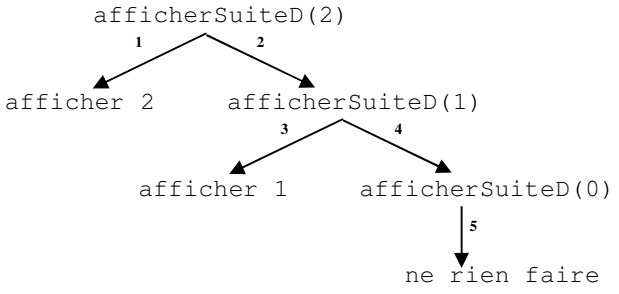
Les programmes récursifs sont souvent concis et élégants mais pas nécessairement efficaces (temps d'exécution et espace mémoire nécessaire) comparativement à une solution itérative.

1.1.2 Bien comprendre le mécanisme de la récursivité

Considérons la procédure récursive suivante :

```
Procédure afficherSuiteD(EntierNaturel n) {
      si n=0 alors ne rien faire ;
      sinon afficher n ;
            afficherSuiteD(n-1) ;
      finSi
finProcédure.
```

Exécutons cette procédure pour `n = 2`. La séquence des traitements réalisés peut être représentée par l'arbre suivant (l'ordre des traitements est indiqué par l'étiquette des arcs) :



Donc `afficherSuiteD(n)` affiche la suite des n premiers entiers par valeurs décroissantes.

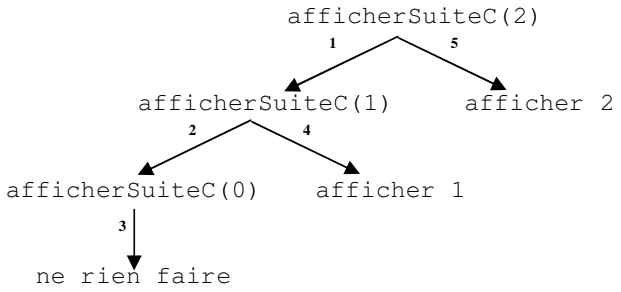
---

[1] Essayer d'exécuter, par exemple, P(1) avec P définie par :
procédure P(entier n) { si n > 0 alors { afficher n ; P(n+1) } }.
Une erreur de débordement de pile traduirait un sous-programme non dérécursivé par le compilateur. En C, compiler par `gcc` en essayant avec ou sans l'option de compilation `-o2`

Intervertissons maintenant l'instruction d'affichage et l'appel récursif :

```
Procédure afficherSuiteC(EntierNaturel n) {
      si n=0 alors ne rien faire ;
      sinon afficherSuiteC(n-1) ;
            afficher n ;
      finSi
finProcédure.
```

Exécutons cette procédure pour `n = 2`. L'ordre des traitements réalisés est représenté par l'arbre suivant :



Donc `afficherSuiteC(n)` affiche la suite des n premiers entiers par valeurs croissantes.

Dans la procédure `afficherSuiteD(n)` les affichages sont ainsi réalisés « à la descente » ; dans la procédure `afficherSuiteC(n)` les affichages sont réalisés « à la remontée ».

L'ordre des appels récursifs est donc important.

1.2 Méthode de conception d'un programme basée sur les invariants

Tout programme P vérifie un énoncé : E {P} S, où E et S sont des propriétés logiques appelées respectivement **pré-condition** et **post-condition**.

Exemples :

$(x \in R)$ { *cos* } ($y = cos(x)$)

$(n>0 \wedge e \in Z)$ { *Rechercher* } ($p = (e \in t[0..n-1])$)

La méthode de conception de programme présentée est synthétisée par le schéma synoptique suivant :

Problème

↓ Spécification formelle

Enoncé E {P} S

↓ Modélisation

Invariant

↓ Construction du programme

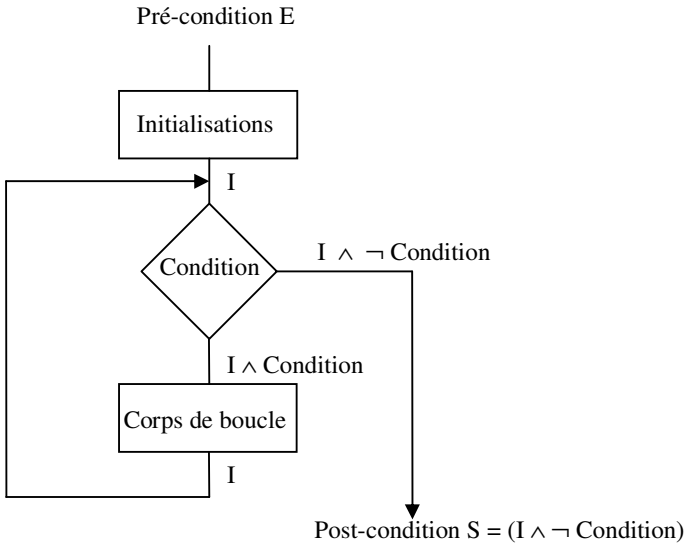
Programme P dont on pourra prouver qu'il vérifie l'énoncé E {P} S

Un **invariant** est une propriété logique sur les valeurs des variables qui caractérise tout ou partie de l'état interne du programme et qui doit être « toujours » vraie. Dans le cas d'un

programme récursif, cet invariant est la relation de récurrence ; dans le cas d'un programme itératif, cet invariant est l'invariant de boucle.

Dans la suite de cette section, on s'intéressera uniquement aux programmes itératifs.

Tout programme itératif $E \{P\} S$ peut se ramener à la structure de programme suivante :



Dans cet organigramme, I représente l'invariant de boucle.

La méthode de construction d'une boucle consiste à :

1) déterminer un invariant I
2) établir la condition d'arrêt de la boucle et vérifier que l'on a bien la propriété :
   (I ∧ Condition d'arrêt) = S
3) construire le corps de boucle :
   a) se rapprocher de la solution
   b) restaurer l'invariant de boucle
4) initialiser les variables et vérifier qu'on a bien :
   E { Initialisations } I

Comment déterminer l'invariant ? Par l'une ou l'autre des grandes approches suivantes (approches strictement équivalentes) :

- Soit paramétrer la post-condition en la généralisant. La question à se poser est : « *Quelle est une situation générale dont la situation finale S n'est qu'un cas particulier ?* » Cette situation générale est l'invariant cherché.
- Soit formuler une hypothèse de récurrence. L'approche est la suivante : « *Supposons qu'on ait déjà réussi à résoudre le même problème pour une taille de problème plus petite* (on ne se pose pas la question du comment on y serait arrivé ! on « suppose que … » !) ». La post-condition de ce problème réduit est l'invariant cherché.

Exemple

Soit à construire un programme P calculant dans une variable s la somme des n éléments d'un tableau t[0..n-1], n>0.

Ce problème peut se spécifier formellement ainsi :

$$(n>0) \{ P \} (s = \sum_{i=0}^{n-1} t[i])$$

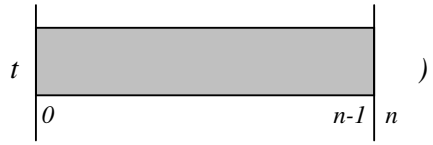
avec $E = (n > 0)$ *et* $S = (s = \sum_{i=0}^{n-1} t[i])$

1) <u>Invariant ?</u>

- *Détermination d'un invariant par la première approche*

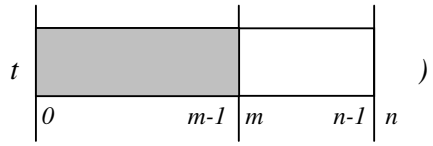
La post-condition S du problème peut s'exprimer de la façon suivante :

S = (s = somme des éléments de la partie grisée du tableau



La situation générale dont la situation finale S n'est qu'un cas particulier est celle dans laquelle s est égale à la somme de m éléments du tableau t[0..n-1], avec m≤n (le cas particulier en question étant le cas m=n). Choisissons, par exemple, les m éléments de t d'indice 0 à m-1 inclus. L'invariant peut s'exprimer ainsi :

I = (s = somme des éléments de la partie grisée du tableau



ou de façon plus formelle : $I(m, s, t) = (s = \sum_{i=0}^{m-1} t[i])$

t et n étant des constantes du problème, s et m étant les seules variables, on allégera un peu l'écriture en écrivant simplement :

$$I(m, s) = (s = \sum_{i=0}^{m-1} t[i])$$

- *Détermination d'un invariant par la deuxième approche (approche équivalente).*

Supposons qu'on ait déjà résolu ce même problème (s = somme des éléments d'un tableau) pour un tableau de taille plus petite que n. Considérons par exemple le sous-tableau t[0..m-1] du tableau t[0..n-1], avec m≤n. Notre hypothèse signifie qu'on suppose avoir déjà résolu le problème suivant :

$$(n>0 \ \wedge \ m \leq n) \{ P \} (s = \sum_{i=0}^{m-1} t[i])$$

Avec ces choix, l'invariant sera :

$$I(m, s) = (s = \sum_{i=0}^{m-1} t[i])$$

2) <u>Condition d'arrêt ?</u>

On arrête l'itération quand m=n, car alors la propriété : $I(m, s) \wedge m=n$ est bien la post-condition S cherchée.

3) <u>Corps de boucle ?</u>

En entrant dans le corps de boucle, on a la propriété : $I(m, s) \wedge m \neq n$.

On se rapproche de la solution cherchée (s = somme des n éléments de t[0..n-1]) en ajoutant au s courant un nouvel élément de t non encore sommé. Le plus simplement accessible est t[m]. L'instruction est : $s \leftarrow s + t[m]$. A l'issue de cette instruction, on a la propriété $I(m+1, s)$.

Il faut maintenant restaurer l'invariant I(m, s). Pour cela, il suffit d'incrémenter m.

Le corps de boucle est donc le suivant (les lignes commençant par // sont des commentaires du programme) :

```
// I(m, s) ∧ m≠n
s ← s + t[m] ;
// I(m+1, s)
m ← m + 1 ;
// I(m, s)
```

4) <u>Initialisations ?</u>

- *Exemple d'initialisations :*

```
// n>0
m ← 1 ;
s ← t[0] ;
// I(m, s)
```

La propriété I(m,s) est bien vérifiée à l'issue de ces initialisations. Elle est liée à la pré-condition n>0 qui assure que t[0] existe.

- *Autre exemple :*

```
// n>0
m ← 0 ;
s ← 0 ;
// I(m, s)
```

Dans ce dernier cas, comment justifier que I(m, s) est bien vérifié ? En effet, l'expression logique

$$s = \sum_{i=0}^{m-1} t[i]$$

n'a pas de sens pour m=0. En fait, le sous-tableau t[0..m-1] n'étant pas défini quand m=0, on peut considérer qu'il ne contient aucun élément et donc que s doit valoir 0.

5) <u>Programme ?</u>

En choisissant par exemple la première version des initialisations, le programme, commenté par son invariant, s'écrit donc :

```
// n > 0
m ← 1 ;
s ← t[0] ;
// I(m, s)
Tant que m≠n faire
```

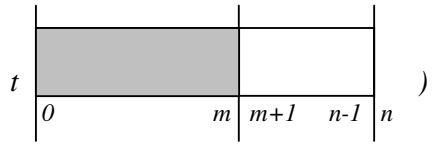
```
        // I(m, s) ∧ m≠n
        s ← s + t[m] ;
        // I(m+1, s)
        m ← m + 1 ;
        // I(m, s)
    FinTantQue
    // I(m, s) ∧ m=n
    //  s = somme des éléments de t[0..n-1]
```

Un même problème peut conduire à plusieurs invariants possibles. Les programmes solutions diffèrent alors quant à leurs initialisations, condition d'arrêt ou/et corps de boucle. Mais ils auront comme caractéristique commune d'être corrects et justifiés.

Exemple

En reprenant le problème de l'exercice précédent, on pourrait envisager d'autres invariants comme par exemple :

- *I2 = (s = somme des éléments de la partie grisée du tableau*



c'est-à-dire : I2(m, s) = ($s = \sum_{i=0}^{m} t[i]$)

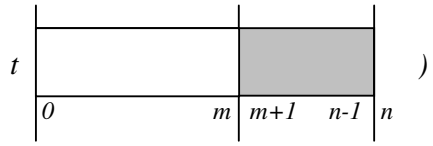
Initialisations : $m \leftarrow 0$; $s \leftarrow t[0]$ // I2(m, s)

Condition d'arrêt : $m = n-1$

Corps de boucle :

```
m ← m+1
// I2(m-1, s)
s ← s + t[m]
// I2(m, s)
```

- *I3 = (s = somme des éléments de la partie grisée du tableau*



c'est-à-dire : I3(m, s) = ($s = \sum_{i=m+1}^{n-1} t[i]$)

Initialisations : $m \leftarrow n-1$; $s \leftarrow 0$ // I3(m, s)

Condition d'arrêt : $m = -1$

Corps de boucle :

```
s ← s + t[m]
// I3(m-1, s)
m ← m-1
// I3(m, s)
```

Commenter un programme par sa pré-condition, son invariant et sa post-condition permet de décrire la sémantique du programme. De tels commentaires fournissent non seulement des éléments de preuve en ligne, mais fournissent aussi des assertions fort utiles pour les tests. Ces principes se retrouvent dans ce qui est appelé la « programmation par contrat ».

1.3 Introduction à la preuve formelle de programme

1.3.1 Le système de Hoare

Un système formel est composé d'axiomes et de règles de déduction. La combinaison de ces règles et axiomes permet d'inférer des théorèmes.

Le système de preuve formelle de programme de Hoare[1] comporte : un seul axiome et des règles de déduction permettant d'inférer des énoncés de la forme E { P } S. Chaque règle de déduction sera écrite sous la forme :

$$p \vdash q$$

où p est la condition d'application de la règle et q la conclusion. Une règle $p \vdash q$ s'interprète comme suit : si p est vrai, alors on peut en déduire q ; ou, de façon équivalente, pour prouver q, il suffit de prouver p. Une règle sans partie gauche ($\vdash q$) peut être appliquée sans condition : c'est un axiome.

Sur le plan sémantique, tous les langages de programmation offrent les mêmes instructions de base :

- l'affection
- les instructions conditionnelles : Si-Alors et Si-Alors-Sinon
- les instructions itératives, qui peuvent toutes se ramener à l'instruction : TantQue-faire
- la composition séquentielle

Le système de Hoare décrit la sémantique de chacune de ces instructions.

Le tableau ci-après liste les axiomes et règles de déduction d'une version simplifiée du système de Hoare. Il fait appel aux notations suivantes : le symbole $\Rightarrow$ représente l'implication logique[2] (ex : $x > 3 \Rightarrow x > 0$) ; les symboles B, E, E', S, S' représentent des expressions logiques ; les symboles P et Q représentent des programmes.

Axiome (règle de l'affectation) $\vdash$ S(…, exp, …) { x $\leftarrow$ exp } S(…, x, …)		
Règles de déduction		
Règle de la post-condition	E{P}S' , S'$\Rightarrow$ S	$\vdash$ E{P}S
Règle de la pré-condition	E $\Rightarrow$ E' , E'{P}S	$\vdash$ E {P} S
Règle du ou	E{P}S , E'{P}S	$\vdash$ (E $\vee$ E') {P} S
Règle du et	E{P}S , E{P}S'	$\vdash$ E {P} (S $\wedge$ S')

[1] Sir Charles Antony Richard HOARE (1934-) est un professeur émérite britannique du Oxford University Computing Laboratory. Il a notamment inventé en 1960 le Quicksort, un des algorithmes de tri les plus performants.

[2] Voir annexes.

Règle du point-virgule	E{P}S' , S'{Q}S	⊢ E {P ; Q} S
Règle du si-alors	E ∧ B {P} S , E ∧ ¬B {} S	⊢ E {Si B alors P}S
Règle du si-alors-sinon	E ∧ B {P} S , E ∧ ¬B {Q} S	⊢ E {Si B alors P sinon Q}S
Règle du tant-que	E ∧ B {P} E	⊢ E {Tant que B faire P} (E ∧ ¬B)

L'axiome d'affectation est en fait un schéma d'axiomes. Il se lit ainsi :

S(…, x, …) est une formule logique quelconque dépendant éventuellement de la variable x

x ← exp est une instruction d'affection de l'expression exp à la variable x

S(…, exp, …) est la formule logique S(…, x, …) dans laquelle toute occurrence éventuelle de la variable x a été remplacée par exp

Cet axiome signifie qu'en toute circonstance, si on a E { x ← exp } S, alors E peut se déduire de S en y remplaçant toutes les occurrences de x par exp. Cet axiome s'utilise habituellement de droite à gauche.

Exemple

Quelle est la pré-condition E de l'énoncé : E { x ← x + 3 } (x > 7) ?

Par application de l'axiome d'affectation, on obtient :

$(x+3 > 7)\ \{ x \leftarrow x + 3 \}\ (x > 7)$

et donc

$(x > 4)\ \{ x \leftarrow x + 3 \}\ (x > 7)$

La réponse est donc : E = (x>4)

Exemple

Quelle est la pré-condition E de l'énoncé :

$E\ \{ x \leftarrow x + y ;\ y \leftarrow x - y ;\ x \leftarrow x - y \}\ (x = a \wedge y = b)$?

En propageant la post-condition de droite à gauche par applications successives de l'axiome d'affection, on obtient successivement

$(x-y = a \wedge y = b)\ \{ x \leftarrow x - y \}\ (x = a \wedge y = b)$

$(x-(x-y)=a \wedge (x-y)=b)\ \{ y \leftarrow x - y \}\ (x-y = a \wedge y = b)$

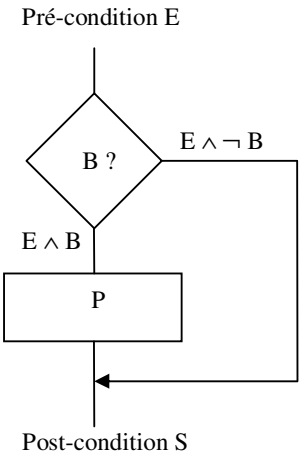
$(y = a \wedge ((x+y)-y) = b)\ \{ x \leftarrow x + y \}\ (y = a \wedge (x-y)=b)$

La réponse est donc : E = (y = a ∧ x = b)

Les 4 premières **règles de déduction** se comprennent facilement dans une lecture gauche-droite. Par exemple, la règle de la pré-condition peut se lire : si on a un énoncé E'{P}S et que, par ailleurs, on peut logiquement déduire E' de E, alors on peut inférer l'énoncé E{P}S.

Les 3 dernières **règles de déduction** se comprennent facilement dans une lecture droite-gauche. Par exemple, la règle du si-alors peut se lire : prouver un énoncé E{Si B alors P}S revient à prouver les deux énoncés E ∧ B {P} S d'une part et E ∧ ¬B {} S d'autre part. La justification sémantique de cette règle s'illustre facilement sur l'organigramme de l'instruction si-alors ci-après.

Quant à l'énoncé de la règle du tant-que, il est à noter que la propriété E représente en fait l'invariant de boucle.



1.3.2 Preuve d'un programme

Prouver un programme, c'est :

1) prouver un énoncé E {P} S
2) montrer que S est atteint en un temps fini

1.3.2.1 Preuve d'un programme itératif

La méthode de preuve consiste à partir de la post-condition et à remonter cette propriété, d'instruction en instruction, en appliquant les règles et axiomes du système de Hoare, jusqu'à retrouver la pré-condition.

Exemple

Reprenons l'exemple du chapitre 1.2 (somme des éléments d'un tableau).

Nous considèrerons que les justificatifs avancés dans la construction du programme et les commentaires insérés dans le programme sont une preuve suffisante de la justesse du programme.

Reste à montrer que S est atteint en un temps fini. La preuve est triviale car : n≥0 par hypothèse ; m=1 par initialisation et est incrémenté de 1 à chaque tour de boucle ; donc m atteindra la valeur d'arrêt n après n-1 itérations.

1.3.2.2 Preuve d'un programme récursif

Prouver un programme récursif, c'est :

1) prouver que le résultat produit dans le cas d'arrêt est correct
2) prouver que les résultats produits dans les cas d'appels récursifs, *sous hypothèse* que ces appels récursifs soient corrects et produisent bien ce qu'il est attendu d'eux, sont corrects
3) prouver la convergence, c'est-à-dire que toute séquence d'appels récursifs conduit toujours à une situation d'arrêt.

Exemple

Prenons l'exemple de la factorielle à récursivité terminale du chapitre 1.1

Dans le cas d'arrêt (n=0 ou n=1), fact renvoie a et donc bien n! = 1.

*Dans le cas de l'appel récursif, et sous hypothèse que l'appel récursif produise le bon résultat, fact renvoie (n-1)! * n * a. Comme a vaut 1, fact renvoie bien n!*

A chaque appel récursif, le premier paramètre est décrémenté de 1. Comme on part de n≥0, fact convergera nécessairement sur la condition d'arrêt n≤1.

1.3.3 En pratique …

La preuve automatique de théorème, et donc d'un énoncé E {P} S, est connue pour être indécidable[1] dans le cas général. Mais s'il s'agit d'un cas décidable, alors il existe un algorithme qui peut déterminer automatiquement, en un temps fini, si un tel énoncé est vrai ou faux.

La preuve formelle complète d'un programme permet d'obtenir une très forte assurance de l'absence de bogue. Toutefois, elle reste souvent coûteuse en temps et en ressources et, en pratique, est généralement réservée aux logiciels les plus critiques[2].

Ceci étant, de façon générale, il est reconnu qu'un processus de conception de programme guidé par les pré-conditions, post-conditions et invariants permet d'obtenir des logiciels de meilleure qualité et d'accélérer les phases de mise au point.

L'approche formelle de la spécification, du développement et de la vérification d'un programme ou d'un système donne lieu à une importante activité de recherche. Parmi les produits, on peut citer par exemple les langages synchrones. Les langages synchrones sont des langages spécialisés conçus pour la programmation sure de systèmes réactifs temps réel. Ils sont utilisés industriellement dans la conception de haut niveau de systèmes complexes (SOC,« System-on-Chip », par exemple) et la programmation de systèmes embarqués. Ces langages adoptent une approche de haut niveau qui s'appuie sur des modèles mathématiques sur lesquels on peut conduire efficacement des analyses et des vérifications de propriétés. Les programmes écrits dans de tels langages se prêtent à des vérifications formelles.

1.4 Efficacité et complexité algorithmique

L'efficacité d'un programme se mesure en terme d'espace mémoire nécessaire et de temps de réponse. Nous nous limiterons à une évaluation de l'efficacité en temps.

Afin de ne pas être dépendant de l'implémentation et de la machine cible, le temps de réponse d'un programme n'est pas évalué en unités de temps mais en nombre d'opérations en fonction de la taille de la donnée à traiter.

1.4.1 Évaluation du temps de calcul

Soit P un programme vérifiant l'énoncé E{P}S. Soit d(n) la donnée de taille n qu'il a à traiter. On dira que P calcule une fonction $f_P(d(n))$ en un temps T(P, d(n)).

Règles d'évaluation du temps de calcul de chaque instruction élémentaire :

---

[1] Voir annexe 11

[2] Un système critique est un système dans lequel les défaillances ont des conséquences graves.

- T(x ← exp, d(n)) = constante pour une instruction simple d'affectation
- T(P;Q , d(n)) = T(P, d(n)) + T(Q , f_P(d(n)))
- T(si B alors P, d(n)) = T_B + si B alors T(P, d(n))

  où T_B est une constante si B est un test simple
- T(si B alors P sinon Q, d(n)) = T_B + si B alors T(P, d(n)) sinon T(Q, d(n))
- T(tant que B faire P, d(n)) = $T_B + \sum_{i=0}^{m-1} (T(P, f_P^i(d(n))) + T_B)$

  $= (m+1)\, T_B + \sum_{i=0}^{m-1} T(P, f_P^i(d(n)))$

  où m est le nombre d'itérations, et $f_P^i(d(n)) = f_P(f_P(\ldots(f_P(d(n))) \ldots))$ i fois.

Exemple

Reprenons l'exemple du chapitre 1.2 (somme des éléments d'un tableau).

		Temps unitaire	*Nb d'exécutions*	*Temps total*
// n > 0 *m ← 1 ;* *s ← t[0] ;* *// I(m, s)*	*initialisations*	*a*	*1*	*a*
Tant que m≠ n faire	*test m≠ n*	*b*	*n*	$\sum_{m=1}^{n} b$
// I(m, s) ∧ m≠n *s ← s + t[m] ;* *// I(m+1, s)* *m ← m + 1 ;* *// I(m, s)*	*corps de boucle*	*c*	*n-1*	$\sum_{m=1}^{n-1} c$
FinTantQue *// I(m, s) ∧ m=n* *// s = somme des élmts de t[0..n-1]*	-	-	-	-

Dans ce cas, d(n) = t[0..n-1] ; n est la taille du problème.

Le programme réalisant n-1 itérations, et chaque instruction élémentaire s'exécutant à temps constant, on a :

$$T(P, t[0..n-1]) = T_{init} + n\, T_{test\ m\neq n} + \sum_{m=1}^{n-1} T_{corps\ de\ boucle}$$

où $T_{init} = T_{m \leftarrow 1} + T_{s \leftarrow t[0]}$ = constante *a*

$T_{test\ m\neq n}$ = constante *b*

$T_{corps\ de\ boucle} = T_{s \leftarrow s+t[m]} + T_{m \leftarrow m+1}$ = constante *c*

Nota. <u>Il est important de souligner que</u> le temps total passé dans le corps de boucle est (n-1) $T_{corps\ de\ boucle}$ *car le temps unitaire* $T_{corps\ de\ boucle}$ *est une constante. Si* $T_{corps\ de\ boucle}$ *avait été dépendant de m, le temps total cumulé n'aurait pas été (n-1)* $T_{corps\ de\ boucle}$ *!!*

In fine, T(P, t[0..n-1]) = $c_1\, n + c_2$ *, où* c_1 *et* c_2 *sont des constantes.*

1.4.2 Complexité algorithmique

La complexité en temps d'un programme définit le comportement asymptotique[1] de son temps de calcul, typiquement soit *en pire cas* soit *en moyenne*, en fonction de la taille du problème :

- un programme P de temps de calcul T(P, d(n)) a une complexité algorithmique en temps en O(f(n)) ssi T(P, d(n)) = O(f(n))
- un programme P de temps de calcul T(P, d(n)) a une complexité algorithmique en temps en θ(f(n)) ssi T(P, d(n)) = θ(f(n))

Dans la notation g(n)=O(f(n)), la borne supérieure asymptotique f(n) peut être très serrée ou très lâche par rapport à g(n). Par exemple, n.log(n)=O(n²) mais également n.log(n)=O(2^n). En pratique, c'est la borne asymptotiquement la plus approchée qui est recherchée pour exprimer la complexité algorithmique. La section 1.4.4.1 définit des complexités de référence usuelles.

Exemple

Reprenons l'exemple précédent.

$T(P, t[0..n-1]) = c_1 n + c_2$, où c_1 et c_2 sont des constantes. Le programme P a donc une complexité en $\theta(n)$.

Exemple

Considérons un polynôme à coefficients réels et de degré r. Les coefficients de ce polynôme sont stockés dans le tableau t[0..r]. Le programme suivant calcule la valeur du polynôme pour $x= x_0$.

```
Procédure EvalPolynome ( entrées : t[0..r], x0 ; sortie : p)
    p ← t[0] ;
    Pour k variant de 1 à r inclus par pas de 1 faire
        x ← x0^k ;
        p ← p + t[k] * x ;
    FinPour
FinProcédure
```

Quelle est la complexité de cette procédure ?

Chaque instruction et opération élémentaire est à temps unitaire constant, sauf éventuellement le calcul de x_0^k. D'où :

$$T(EvalPolynome, t[0..r]) = T_{passage\ paramètres} + T_{init\ p} + T_{init\ k} + T_{test\ k\le r} + \sum_{k=1}^{r} (T_{x \leftarrow x0^k} + T_{p \leftarrow p+t[k]*x} + T_{k \leftarrow k+1} + T_{test\ k\le r}) + T_{retour\ du\ résultat}$$

$$= a + b\,r + \sum_{k=1}^{r} T_{x0^k} \qquad \text{où a et b sont des constantes}$$

- *Si x_0^k est calculée par une boucle spécifique :*

```
x ← 1 ;
Pour i variant de 1 à k inclus, par pas de 1, faire x ← x * x0  FinPour
```

[1] Voir définition des notations O et θ en annexe 8.2

alors le temps de calcul unitaire de x_0^k sera : $T_{x0}{}^k = c + d\,k$, où c et d sont des constantes. Le temps de calcul total de la procédure sera donc :

$$T(EvalPolynome, t[0..r]) = a + b\,r + \sum_{k=1}^{r} (c + d\,k)$$
$$= a + (b+c)r + d\,r\,(r+1)/2$$
$$= \theta(r^2)$$

La complexité en temps du programme sera donc en $\theta(r^2)$.

- *Si, par contre, x_0^k est calculée sans boucle spécifique mais à partir de la valeur x_0^{k-1} mémorisée de l'itération précédente, alors une seule multiplication par x_0 suffira pour calculer la valeur courante de x_0^k. Le temps unitaire de calcul de x_0^k sera : $T_{x0}{}^k$ = une constante c' indépendante de k. Le temps de calcul total de la procédure sera donc :*

$$T(EvalPolynome, t[0..r]) = a' + b\,r + \sum_{k=1}^{r} c'$$
$$= \theta(r)$$

La complexité en temps du programme sera donc en $\theta(r)$.

Le calcul de la complexité d'un algorithme n'est pas toujours chose triviale et ne conduit évidemment pas toujours à une expression simple en fonction de la taille n du problème. *Exemples d'algorithmes performants : multiplication de grands nombres[1] en $O(n.\log(n).\log(\log(n)))$; multiplication de deux matrices carrées[2] en $O(n^{2.376})$.*

1.4.3 Calcul de complexité d'un programme récursif

Le temps de calcul des programmes récursifs s'exprime très naturellement par une relation de récurrence. Voici quelques relations types courantes et la complexité qui s'en déduit.

Expression du temps de calcul (a, b, et c étant des constantes)	Complexité résultante
$T(n=0) = a$ $T(n>1) = b + T(n-1)$	$\theta(n)$
$T(n=1) = a$ $T(n=2^{p>0}) = b + T(n/2)$	$\theta(\log(n))$
$T(n=1) = a$ $T(n=2^{p>0}) = b + c\,n + 2\,T(n/2)$	$\theta(n.\log(n))$

Le chapitre 10 fournit des techniques pratiques de résolution de telles récurrences.

---

[1] SCHÖNHAGE, A., STRASSEN, V. « Schennelle Multiplikation grosser Zahlen », *Computing*, vol. 7, 1971, p. 281-292

[2] COPPERSMITH, D., WINOGRAD, S. « Matrix multiplication via arithmetic progressions », *19th Annual ACM Symposium on Theory of Computing*, 1987, p.1-6

1.4.4 Comparaison d'efficacité

1.4.4.1 Grandes classes d'algorithmes

Exemples de grandes classes usuelles d'algorithmes, listées par complexité croissante en fonction de la taille n du problème à traiter :

O(1)	Exprime un temps de calcul borné par une constante
O(log(log(n)))	Ex : complexité en moyenne d'une recherche par interpolation
O(log(n))	Complexité logarithmique (ex : recherche dichotomique)
O(n)	Complexité linéaire (ex : recherche séquentielle)
O(n.log(n))	Complexité linéaire-logarithmique (ex : bon algorithme de tri)
$O(n^2)$	Complexité quadratique (ex : tri à bulles)
$O(n^{constante})$	Complexité polynomiale
$O(2^n)$	Complexité exponentielle
O(n!)	Complexité factorielle

Exemple en chiffres.

Une page A4 représente $\frac{1}{16}m^2$ à raison de 300 points par pouce, soit 8718750 points.
Avec n=8718750 :

- *un algorithme en n^2 nécessitera* $7.6\ 10^{13}$ *opérations*
- *un algorithme en $n.log_2(n)$ nécessitera* $2.0\ 10^{8}$ *opérations*
- *un algorithme en n nécessitera* $8.7\ 10^{6}$ *opérations*
- *un algorithme en $log_2(n)$ nécessitera* 23 *opérations !*

On appelle **algorithmes polynomiaux** ceux dont la complexité est en O(f(n)), où f(n) est un polynôme. On appelle **algorithmes non-polynomiaux** ceux dont la complexité est exponentielle ou factorielle. En pratique, on considère que seuls les algorithmes polynomiaux sont performants (même si les algorithmes en $O(n^{k\geq 3})$ sont considérés comme lent), les algorithmes non-polynomiaux étant impraticables dès que n dépasse quelques dizaines.

Exemple illustrant la rupture entre polynomial et non polynomial

Soit un calculateur réalisant une « opération » par microseconde. Le tableau ci-après détermine le temps de calcul d'un algorithme de f(n) « opérations » en fonction de la taille n du problème.

	n = 10	***n = 20***	***n = 40***	***n = 60***
$f(n) = n^2$	***10^{-4} s***	***$4*10^{-4}$ s***	***$1.6*10^{-3}$ s***	***$3.6*10^{-3}$ s***
$f(n) = n^3$	***10^{-3} s***	***$8*10^{-3}$ s***	***$6.4*10^{-2}$ s***	***$2.2*10^{-1}$ s***
$f(n) = 2^n$	***10^{-3} s***	***1 s***	***12.7 jours***	***366 siècles***
f(n) = n !	***3.6 s***	***100 siècles***	***…***	***…***

Il est important de souligner que la rupture entre polynomial et non polynomial ne se réduira pas avec l'évolution technologique ! Plus l'ordinateur cible sera rapide, plus un programme de faible complexité algorithmique sera intéressant !

Exemple

Pour un problème qui nécessite f(n) opérations, quelle est la taille n maximale du problème résolvable en un temps donné sur une machine de puissance donnée ?

	En 1000 s sur une machine X réalisant une « opération » par seconde	***En 1000 s sur une machine Y 1000 fois plus rapide que la machine X***	**Gain relatif**
$f(n) = 100\ n$	$n_{max} = 10$	$n_{max} = 10000$	*1000*
$f(n) = 5\ n^2$	$n_{max} = 14$	$n_{max} = 447$	*32*
$f(n) = n^3 / 2$	$n_{max} = 12$	$n_{max} = 126$	*10*
$f(n) = 2^n$	$n_{max} = 10$	$n_{max} = 20$	*2*

1.4.4.2 Efficacité réelle

Deux algorithmes résolvant un même problème et de même complexité sont généralement, en première approche, considérés de même efficacité. Toutefois, pour une comparaison réelle d'efficacité, il ne suffit pas de comparer uniquement les complexités (qui sont des temps de calcul asymptotiques) : il faut aussi prendre en compte les valeurs des constantes cachées derrière les notations O ou θ !

Exemple

Si $T(P_1, d(n)) = 10\ n^2$, alors la complexité de P_1 est en $\theta(n^2)$

Si $T(P_2, d(n)) = 1000\ n$, alors la complexité P_2 en $\theta(n)$

Il s'en suit que le programme P_2 est plus efficace que le programme P_1 … mais uniquement si n est suffisamment grand. C'est le contraire si $n < 100$.

1.4.4.3 Complexité d'algorithmes parallèles

Nous avons toujours supposé jusqu'à présent que les algorithmes développés étaient séquentiels et exécutés sur une machine monoprocesseur. Mais si l'algorithme se prête à la parallélisation et si l'on dispose d'une machine multiprocesseurs permettant d'exécuter des opérations en parallèle, alors on peut prétendre à gagner en temps de calcul.

Exemple

Le calcul de n! peut s'exécuter sur une machine de n processeurs élémentaires en $\theta(log(n))$ opérations. En effet :

$$\forall n>1 \quad n! = \prod_{i=1}^{n\,div\,2} i \quad * \prod_{i=(n\,div\,2)+1}^{n} i$$

Les deux termes de la multiplication peuvent être exécutés en parallèle, chacun sur un processeur distinct. Et ainsi à chaque étape. Le temps de calcul de la fonction fact est ainsi déterminé par la relation de récurrence suivante :

$$T(fact, 0) = T(fact, 1) = \text{constante } a$$

$$\forall n>1 \quad T(fact, n) = T(fact, n/2) + T_*$$

En prenant $n=2^k$, on obtient : $T(fact, 2^k) = a + b\ k = \theta(log(n))$.

2 THÉORIE DE LA COMPLEXITE – CLASSES DE PROBLÈMES

La théorie de la complexité s'intéresse à l'étude formelle de la difficulté des problèmes en informatique, la question étant de savoir si ces problèmes peuvent être résolus efficacement ou pas en se basant sur une estimation théorique des temps de calcul et des besoins en espace mémoire.

On distingue deux grands types de problèmes :

- les *problèmes de décision* : ce sont les problèmes posant une question dont la réponse est oui ou non ;
- les *problèmes d'optimisation* : ce sont les problèmes cherchant à optimiser une certaine valeur.

Exemple

> *Considérons « le problème du voyageur de commerce ». Etant donné un ensemble de n villes et pour chaque paire de villes (v_i , v_j) la distance de la ville v_i à la ville v_j :*
>
> *- version 1 : trouver un parcours fermé passant par toutes les villes et dont la longueur est inférieure à une constante donnée L ;*
>
> *- version 2 : trouver un parcours fermé passant par toutes les villes et qui minimise la distance parcourue.*
>
> *La version 1 est un problème de décision ; la version 2 est un problème d'optimisation. A noter qu'un algorithme résolvant le problème d'optimisation résout le problème de décision (il suffit de comparer la distance minimale à L).*

La théorie de la complexité ne traite fondamentalement que des problèmes de décision. Cependant on étend la notion de complexité aux problèmes d'optimisation. En effet il est facile de transformer un problème d'optimisation en problème de décision. Par exemple, chercher à optimiser une certaine valeur n revient à traiter un problème de décision qui consiste à comparer n à un certain k : en générant un certain nombre de valeurs k on peut déterminer la valeur optimale cherchée.

On dit qu'un problème est **facile** s'il existe un algorithme de complexité polynomiale le résolvant ; on dit qu'un problème est **difficile** s'il n'existe pas d'algorithme de complexité polynomiale le résolvant.

Exemples

> *Déterminer l'enveloppe convexe d'un nuage de points est un problème facile (l'algorithme de Graham est en O(n.log(n)), et donc en $O(n^2)$) ; déterminer le plus court chemin dans un graphe est un problème facile (l'algorithme de Floyd est en $O(n^3)$) ; le problème du voyageur de commerce est difficile (pas d'algorithme connu de complexité polynomiale).*

Si un problème d'optimisation est *simple*, alors son problème de décision associé est *simple* également ; si un problème de décision est *difficile*, alors le problème d'optimisation associé est *difficile* aussi. En pratique, on confond donc souvent un problème d'optimisation et son problème de décision associé.

La ***complexité d'un problème*** est la complexité du meilleur algorithme existant qui résout ce problème. La notion de ***classe de complexité*** *(classe de problèmes)* permet de classer les problèmes en fonction leur complexité.

2.1 Algorithmes déterministes et non-déterministes

La définition des classes de problèmes fait appel aux notions d'algorithmes déterministe et non-déterministe :

- un ***algorithme déterministe*** est un algorithme classique qui fait des choix déterministes à chaque étape ;
- un ***algorithme non-déterministe*** est un algorithme <u>*théorique*</u> qui est doté d'un *oracle* lui permettant, lorsque l'algorithme se trouve devant plusieurs voies possibles (pensez par exemple à la recherche d'un chemin dans un graphe), de choisir systématiquement la meilleure. Un *oracle* peut être vu comme une fonction en O(1) qui répond à une question difficile.

  A noter que tout algorithme déterministe est un cas particulier d'algorithme non-déterministe.

En pratique, les algorithmes que l'on programme sont tous déterministes, car on ne sait pas construire de machine non-déterministe. On ne peut donc que *simuler* un algorithme non-déterministe par un algorithme déterministe. Une façon d'implémenter un algorithme non-déterministe est d'utiliser le « *backtracking* » ou d'explorer toutes les solutions possibles en parallèle.

> Le « ***backtracking*** » consiste, face à un choix, à faire une supposition de voie à suivre et à poursuivre la recherche de la solution, puis, si on se retrouve dans l'impossibilité de résoudre le problème, à revenir en arrière et changer la supposition. Le backtracking est l'implémentation d'une stratégie de résolution par essais-erreurs, même si les suppositions peuvent être éclairées par des heuristiques[1].

Il est démontré qu'un algorithme déterministe qui simule un algorithme intrinsèquement non-déterministe s'exécutant en temps polynomial, fonctionne en temps exponentiel.

2.2 Classes de complexité de problème

Les trois classes de complexité de problème les plus courantes sont P, NP, NP-Complet.

2.2.1 Classe P

Un problème de décision est dans P s'il peut être résolu par un algorithme déterministe en un temps *polynomial* par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.

La classe P est la classe des problèmes *faciles*.

<u>Retenir</u> : la classe P est formée des problèmes qui peuvent être résolus en temps polynomial. Etre dans P c'est « <u>trouver</u> une solution en temps polynomial ».

[1] Une heuristique est une règle empirique guidant une recherche.

2.2.2 Classe NP

La classe NP est la classe des problèmes de décision pour lesquels la réponse *oui* peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance. NP signifie « *non-déterministe polynomial* » (et non pas non-polynomial !!!).

De manière intuitive, dire qu'un problème peut être décidé à l'aide d'un algorithme non-déterministe polynomial signifie que :

- il est facile, pour une solution donnée, de vérifier par un algorithme déterministe en un temps polynomial si celle-ci répond au problème pour une instance donnée, …
- … mais que le nombre de solutions à tester pour résoudre le problème peut très bien être exponentiel par rapport à la taille de l'instance.

Les problèmes NP sont ceux qui peuvent être résolus en supposant une solution puis en vérifiant la supposition en temps polynomial. La classe NP est donc composée de problèmes pour lesquels une solution peut être rapidement validée.

Exemple

> *Considérons le problème de la factorisation d'un entier en produit de facteurs premiers. On ne sait pas s'il existe un algorithme polynomial qui sache le résoudre. On ne sait donc pas si ce problème est dans P. Par contre, étant donné k nombres p_1, p_2, … p_k, il est trivial de vérifier si $n = p_1 \times p_2 \times \ldots \times p_k$. Ce problème est donc dans NP.*

A noter que tout problème facile (c.-à-d. dans P) est trivialement dans NP ; par contre un problème difficile (au sens défini antérieurement) peut ne pas être dans NP.

Retenir : les problèmes NP sont ceux pour lesquels *une solution* peut-être validée en temps polynomial. Etre dans NP c'est « prouver en temps polynomial qu'une proposition de réponse est solution du problème ».

2.2.3 Classe NP-Complet

La classe NP-Complet est composée des problèmes les plus *difficiles* de NP. C'est le noyau dur des problèmes de NP. On dit qu'un problème est NP-Complet si sa résolution en temps polynomial entraînerait la résolution en temps polynomial de tout problème de NP.

Une définition un peu plus formelle est donnée en annexe 11.

Retenir : les problèmes NP-Complets sont les problèmes de NP que personne ne sait résoudre plus vite qu'en temps exponentiel.

2.2.4 Conjecture P ≠ NP

On sait que $\mathbf{P \subseteq NP}$. Mais on ne sait pas si P égale ou non NP. S'il s'avérait que P = NP, alors on pourrait résoudre tous les problèmes NP en un temps polynomial sur une machine déterministe. Or les problèmes NP-Complets sont très fréquents et pour aucun d'eux on n'a réussi à trouver un algorithme polynomial le résolvant. De plus, intuitivement, on aurait tendance à penser que la recherche d'une solution prend plus de temps que de vérifier l'exactitude de la solution, et que donc NP serait plus large que P. En général, la communauté

scientifique pense que P ≠ NP, mais ce n'est pas prouvé et le problème, posé depuis 1971, reste ouvert. La **conjecture P ≠ NP** est un problème très fondamental en informatique théorique et a une portée pratique considérable (par exemple c'est sur cette conjecture que repose quasiment toute la cryptographie). Ce problème est si important qu'il fait partie des 7 problèmes du millénaire, dont la résolution est primée 1 million de dollars par le Clay Mathematic Institute[1].

2.3 Problèmes NP-Complets

Beaucoup de problèmes importants et fréquents sont NP-Complets. Par exemples :

- le problème du voyageur de commerce : étant donné *n* villes et les distances séparant chaque ville deux à deux, trouver un parcours fermé de longueur minimale qui passe par toutes les villes.
- le problème de l'isomorphisme de graphes : étant donné deux graphes G1 et G2, déterminer si G1 est isomorphe à un sous-graphe de G2.
- le problème de la satisfiabilité d'une formule logique : étant donné une expression booléenne F de *m* clauses sur *n* variables, existe-t-il une façon d'assigner les valeurs « vrai » ou « faux » aux variables afin de rendre l'expression F vraie ?
- le problème de la partition d'un ensemble : peut-on diviser un ensemble d'entiers en deux ensembles de même somme ?
- le problème du sac à dos : étant donné un ensemble E d'entiers naturels et t un entier naturel, existe-t-il un sous-ensemble E' de E dont les éléments ont pour somme t ?
- le problème du placement-routage : sur une surface donnée, quel agencement permet de placer un maximum d'objets de tailles et formes données.
- le problème de la coloration de graphe : quelle est le nombre minimal de couleurs permettant d'attribuer une couleur à chaque sommet d'un graphe sans que deux sommets adjacents soient de la même couleur ?

On peut considérer les problèmes NP-Complets comme quasi-certainement intraitables avec efficacité. Que faire alors face à un problème NP-Complet ? Plusieurs approches sont possibles :

- des *algorithmes d'approximation* permettent de trouver des solutions approchées de l'optimum en un temps raisonnable ;
- des *algorithmes stochastiques* : en utilisant des nombres aléatoires on peut «forcer» l'algorithme à ne pas utiliser les cas les moins favorables ;
- des *heuristiques* permettent d'obtenir des solutions généralement bonnes, mais non exactes, en un temps de calcul modéré ;
- des algorithmes par *séparation et évaluation* permettent de trouver la ou les solutions exactes. Bien qu'en temps de calcul évidemment non polynomial, il peut rester modéré pour certaines classes de problèmes et des instances relativement grandes ;
- on peut aussi restreindre la classe des problèmes d'entrée à une sous-classe suffisante mais plus facile à résoudre.

Si ces approches échouent, le problème est non soluble en pratique (pour des entrées de grande taille quelle que soit la puissance de la machine) dans l'état actuel des connaissances.

---

[1] CLAY MATHEMATICS INSTITUTE. *P vs NP Problem.* www.claymatth.org/millenium/P_vs_NP/

3 ALGORITHMES DE RECHERCHE

On se pose le problème de rechercher un élément ou le nombre d'occurrence d'un élément dans un ensemble d'éléments organisé linéairement en mémoire centrale (tableau ou liste).

Nota. La recherche sur un support externe (disque) est hors du champ couvert ici, bien que d'importance (recherche dans un grand fichier, dans une base de données, …). Des contraintes spécifiques sont liées à ce type de problème : l'ensemble des données est souvent trop volumineux pour pouvoir être totalement chargé en mémoire centrale, et il faut minimiser le nombre d'accès disque, car ceux-ci représentent le temps d'exécution dominant. Les solutions mettent en œuvre des techniques spécifiques (B-Arbres ou hachage étendu, par exemple).

3.1 Recherche séquentielle

Soit un tableau t[0..n-1]. Soit e une valeur du type des éléments de t. Construire un algorithme vérifiant l'énoncé :

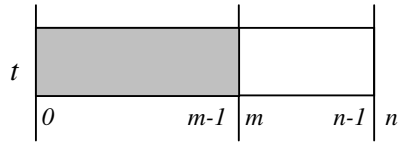
(n>0) { P } (trouvé = e∈ t[0..n-1])

L'algorithme P peut s'écrire :

// n>0
P_1 ;
// t[0..m-1] est le sous-tableau de taille maximale commençant en 0 tel que e∉ t[0..m-1]
trouvé ← (m≠n) ;
// trouvé = e∈ t[0..n-1]

Résolvons P_1.

Invariant possible : I(m) = (e∉ t[0..m-1])



Condition d'arrêt : (m=n) ∨ (t[m]=e)
car alors la post-condition de P_1 est vérifiée.

Corps de boucle : // I(m) ∧ m≠n ∧ (t[m]≠e)
m ← m+1 ;
// I(m)

Initialisations : // (n>0)
m ← 0 ;
// I(m)

L'algorithme P s'écrit donc :

```
// « Recherche séquentielle » de e dans t[0..n-1], avec n>0.
m ← 0 ;
// I(m) = ( e∉t[0..m-1] )
tant que (m≠n) et (t[m]≠e) faire
      // I(m) ∧ m<n ∧ (t[m]≠e)
      m ← m+1 ;
      // I(m)
finTantQue
// t[0..m-1] est le sous-tableau de taille maximale commençant en 0
// et tel que e∉t[0..m-1]
trouvé ← (m≠n) ;
// trouvé = e∈t[0..n-1]
```

Complexité algorithmique : O(n).

3.2 Recherche dichotomique

t[0..n-1] est maintenant supposé trié par valeurs strictement croissantes. On notera : t[0..n-1]↑

Construire un algorithme vérifiant l'énoncé :

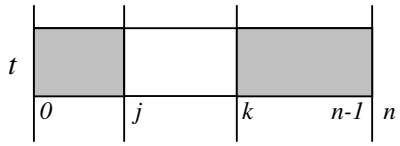
((n>0) ∧ (t[0..n-1]↑)) { P } (si e∈t[0..n-1] alors t[j]=e sinon j=-1)

L'algorithme P peut s'écrire :

```
// (n>0) ∧ (t[0..n-1]↑)
si (e<t[0]) ou (e>t[n-1]) alors j ← -1 ;
sinon
      // (n>0) ∧ (t[0..n-1]↑) ∧ (t[0]≤e≤t[n-1])
      P1 ;
      // t[j]≤e<t[j+1]
      si t[j]≠e alors j ← -1 finsi ;
finSi
// si e∈t[0..n-1] alors t[j]=e sinon j=-1
```

Résolvons P_1.

Invariant possible : I(j, k) = (t[j] ≤ e < t[k])



Condition d'arrêt : j = k-1
car alors la post-condition de P_1 est vérifiée.

Corps de boucle :

La propriété d'ordre sur les éléments de t permet de réduire la plage d'indices [j, k[de moitié à chaque itération car, en notant m = (j+k)/2 :

1er cas : (I(j, k) ∧ t[j]≤e<t[m]) ⇒ I(j, m)

2ème cas : (I(j, k) ∧ t[m]≤e<t[k]) ⇒ I(m, k)

Initialisations : // $(n>0) \wedge (t[0..n-1]\uparrow) \wedge (t[0] \leq e \leq t[n-1])$
j ← 0 ;
k ← n ;
// I(m)

Il y a deux façons de justifier la propriété I(m) à l'issue des initialisations. La première est de considérer que, sur le plan logique, le tableau t[0..n-1] étant croissant, on peut prendre une définition étendue de t telle que $t[n]=+\infty$ (considération conceptuelle, pas physique!). La seconde, plus formelle, est de se rappeler que le véritable énoncé de l'invariant est :

$I(j, k) = ((0 \leq j < k \leq n-1) \Rightarrow (t[j] \leq e < t[k]))$

où $\Rightarrow$ dénote l'implication logique, et donc que I(j, k) est vrai si k=n.

L'algorithme P s'écrit donc :

```
// « Recherche dichotomique » de e dans t[0..n-1]↑, avec n>0.

si (e<t[0]) ou (e>t[n-1]) alors j ← -1 ;
sinon
      // (n>0) ∧ (t[0..n-1]↑) ∧ (t[0]≤e≤t[n-1])
      j ← 0 ;
      k ← n ;
      // I(j,k) = ( t[j]≤e<t[k] )
      tant que (j≠k-1) faire
            // I(j,k) ∧ j<k-1
            m ← (j+k)/2 ;
            si ( e<t[m] )
                  alors // I(j,m)
                        k ← m ;
                        // I(j,k)
                  sinon // I(m,k)
                        j ← m ;
                        // I(j,k)
            finSi ;
            // I(j,k)
      finTantQue ;
      // I(j,j+1)
      si ( t[j]≠e ) alors j ← -1 finsi ;
finSi ;
// si e∈t[0..n-1] alors t[j]=e sinon j=-1
```

Quelle est la complexité de cet algorithme ?

La plage de recherche [j, k[est divisée par deux à chaque itération jusqu'à ce qu'elle se réduise à un singleton. Combien de fois le corps de boucle sera-t-il exécuté ? Supposons d'abord $n = 2^p$, avec $p \in N^*$: le corps de boucle s'exécutera p fois ; supposons maintenant $2^p < n < 2^{p+1}$: le corps de boucle s'exécutera au pire p+1 fois. Le temps d'exécution du programme sera donc, en pire cas, de la forme $T(P, t[0..n-1]) = a + b\,p$, où a et b sont des constantes et $p = \lfloor \log_2(n) \rfloor$. L'algorithme a donc une complexité en O(log(n)).

Remarques :

- Ne pourrait-on pas arrêter la boucle dès que e=t[j] ? Pour cela il faudrait réaliser un test supplémentaire à chaque tour de boucle, mais pour quel enjeu ? Le nombre d'itérations étant très faible (par exemple, 20 itérations au pire pour un tableau de 10^6 éléments), le gain sur le nombre d'itérations pourrait facilement, en moyenne, ne pas compenser le coût supplémentaire de ce test.
- On a supposé initialement t[0..n-1] strictement croissant. Mais on peut vérifier que cet algorithme fonctionne également si t comporte des répétitions d'éléments.

3.3 Recherche dans une matrice bi-ordonnée

Soit t[0..m-1, 0..n-1] une matrice de m lignes et n colonnes, strictement croissante en ligne et strictement croissante en colonne.

Exemple

m-1=2	*16*	*25*	*30*	*40*	*47*
1	*5*	*20*	*25*	*30*	*32*
0	*2*	*16*	*20*	*25*	*26*
	0	*1*	*2*	*3*	*4=n-1*

On cherche le nombre d'occurrences d'un élément e dans t[0..m-1, 0..n-1] :

- Solution 1. On réalise une recherche séquentielle en O(n) sur chacune des m lignes. L'algorithme sera en O(m.n), donc de nature quadratique.
- Solution 2. On réalise une recherche dichotomique en O(log(n)) sur chacune des m lignes. L'algorithme sera en O(m.log(n)), donc linéaire-logarithmique.
- Solution 3. On souhaite un algorithme en O(m+n), donc linéaire.

Soit donc à construire un algorithme en O(m+n) vérifiant l'énoncé :

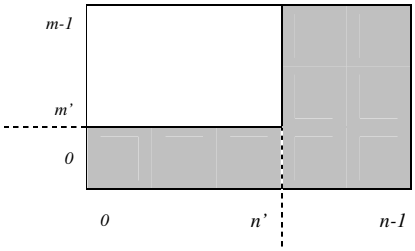
((m>0) ∧ (n>0) ∧ t[0..m-1, 0..n-1] ↑en ligne et ↑en colonne)

{ P }

(k = #(e, t[0..m-1, 0..n-1]))

où #(e, t[0..m-1, 0..n-1]) est le nombre d'occurrences de e dans la matrice t[0..m-1, 0..n-1].

Invariant possible : I(k, m', n') = (k = nombre d'occurrences de e dans la partie grisée de la matrice t)



Ce qui peut s'écrire, par exemple :

I(k, m', n') = (k = #(e, t[0..m'-1, 0..n-1]) + #(e, t[m'..m-1, n'+1..n-1]))

Ou encore, de façon équivalente :

I(k, m', n') = (#(e, t[0..m-1, 0..n-1]) = k + #(e, t[m'..m-1, 0..n']))

Nota : le choix de la partie grisée n'est pas unique mais pas indépendante de l'ordre des éléments en ligne et en colonne !

Condition d'arrêt : m'= m ∨ n'= -1
car alors la post-condition de P est vérifiée.

Corps de boucle :

1er cas : (I(k, m', n') ∧ t[m', n']=e) ⇒ I(k+1, m'+1, n'-1)
2ème cas: (I(k, m', n') ∧ t[m', n']<e) ⇒ I(k, m'+1, n')
3ème cas: (I(k, m', n') ∧ t[m', n']>e) ⇒ I(k, m', n'-1)

Initialisations : // (m>0) ∧ (n>0) ∧ (t[0..m-1,0..n-1] ↑en ligne et ↑en colonne)
m' ← 0 ;
n' ← n-1 ;
k ← 0 ;
// I(k, m', n')

La justification de la propriété I(k, m', n') à l'issue des initialisations se réalise d'une façon similaire à celle détaillée pour l'algorithme de recherche dichotomique de la section précédente.

L'algorithme P s'écrit donc :

```
// « Recherche dans une matrice bi-ordonnée ».
// recherche de e dans t[0..m-1, 0..n-1] ↑en ligne et ↑en
// colonne, avec m>0, n>0.

// (m>0) ∧ (n>0) ∧ (t[0..m-1, 0..n-1] ↑en ligne et ↑en colonne
m' ← 0 ;
n' ← n-1 ;
k  ← 0 ;
// I(k, m', n') = ( k = #(e, t[0..m'-1, 0..n-1]) +
//                    #(e, t[m'..m-1, n'+1..n-1]) )
tant que (m'<m) ∧ (n'>-1) faire
      // I(k, m', n') ∧ (m'<m) ∧ (n'>-1)
      si ( t[m',n']=e ) alors
            // I(k+1, m'+1, n'-1)
            k ← k+1 ; m' ← m'+1 ; n' ← n'-1 ;
            // I(k, m', n')
      sinon si ( t[m',n']<e ) alors
            // I(k+1, m'+1, n')
            m' ← m'+1 ;
            // I(k, m', n')
      sinon // I(k+1, m', n'-1)
            n' ← n'-1 ;
            // I(k, m', n')
      finSi ;
      // I(k, m', n')
finTantQue ;
// k = #(e, t[0..m-1, 0..n-1])
```

Quelle est la complexité de cet algorithme ? La propriété d'ordre sur les éléments de t permet au corps de boucle d'éliminer à chaque itération toute une ligne ou/et toute un colonne. Le

nombre maximal d'itérations est obtenu quand une seule ligne ou colonne est éliminée à chaque itération, ce qui conduit à un total en pire cas de m+n itérations. Chacune des opérations élémentaires étant en θ(1), l'algorithme P a donc une complexité en O(m+n).

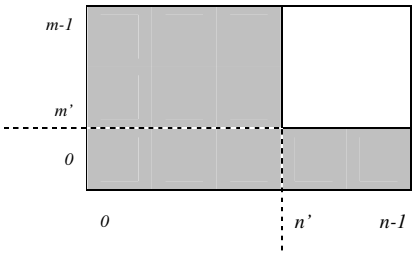
Nota. La recherche dans une matrice bi-ordonnée est un problème générique, car de nombreux problèmes peuvent s'y ramener, la matrice t n'étant pas alors nécessairement physiquement construite mais guidant le raisonnement.

Exemple

Soit E1[0..m-1] et E2[0..n-1] deux ensembles triés strictement croissants. La matrice t[0..m-1, 0..n-1] de terme général t[i,j]=E1[i]-E2[j] est strictement croissante en colonne et strictement décroissante en ligne. Les zéros de t[i,j] identifient les éléments de l'intersection ensembliste E1∩E2. Déterminer le cardinal de E1∩E2 revient à déterminer le nombre de 0 dans la matrice (virtuelle, car non construite !) t ; déterminer l'intersection E1∩E2 revient à déterminer tous les couples (i,j) tels que t[i,j]=0. Le problème se ramène donc à une recherche dans une matrice (virtuelle) bi-ordonnée. Compte tenu des propriétés d'ordre des éléments de t, l'invariant pour la détermination du cardinal E1∩E2, par exemple, pourrait être :

I(k, m', n') = (k = nombre de 0 dans la partie grisée de la matrice t)

= (k = cardinal(E1[0..m-1] ∩ E2[0..n-1]) –
cardinal(E1[m'..m-1] ∩ E2[n'..n-1]))



Quand la condition d'arrêt m'=m ou n'=n est atteinte, k = cardinal(E1 ∩ E2).

4 REPRÉSENTATION D'UN ENSEMBLE

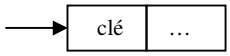
4.1 Ensembles dynamiques – Dictionnaires

Les ensembles manipulés par des programmes peuvent croître, diminuer ou subir d'autres modifications au cours du temps. On dit que ce sont des **ensembles dynamiques**.

Chaque élément d'un ensemble dynamique est typiquement représenté par un objet auquel on accède via un pointeur et dont les champs peuvent être examinés et manipulés. Généralement, un tel objet contient deux grandes informations :

- une clé, qui sert d'identifiant de l'élément (toutes les clés ne sont pas obligatoirement distinctes)
- des données satellites, stockées dans d'autres champs

Par la suite, on représentera graphiquement un tel objet de la façon suivante :



le champ de droite « … » représentant les données satellites.

Un **dictionnaire** est un ensemble dynamique qui supporte les opérations suivantes :

rechercher(clé)[1], insérer(élément), supprimer(élément)

Un ensemble dynamique, s'il est totalement ordonné par rapport à ses clés, supporte aussi, classiquement, les opérations suivantes :

minimum()[1,2], maximum()[1,2], successeur(clé) [1], prédécesseur(clé) [1].

Nota : [1] ces fonctions renvoient un pointeur sur l'élément cherché.
[2] ces fonctions renvoient l'élément de clé extrémale

Dans la suite de ce chapitre, on se restreint à des dictionnaires.

4.2 Tables

4.2.1 Tables à adressage direct

Soit U = {0, 1, 2, …, M-1} l'univers des clés possibles.

Soit K = l'univers des clés réellement stockées dans le dictionnaire considéré. $K \subseteq U$.

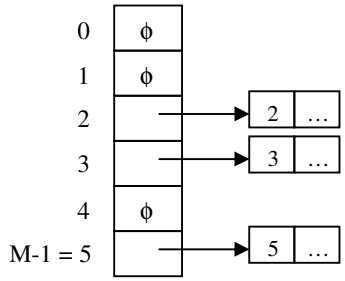
Supposons :

- M pas trop grand
- chaque élément de l'ensemble possède une clé distinctive unique (toutes les clés du dictionnaire sont distinctes)

Le dictionnaire peut être représenté par un tableau t[0..M-1] dans lequel chaque case – alvéole – est destinée à stocker un élément : l'élément de clé k est stocké dans l'alvéole d'indice k.

Exemple

En supposant U = {0, 1, 2, 3, 4, 5} et K = {2, 3, 5}, le dictionnaire serait :



où le symbole ϕ représente un pointeur nul.

Avantages :

- chaque opération rechercher, insérer et supprimer est en θ(1)
- s'il n'y a pas de données satellites, le dictionnaire peut se réduire à un simple tableau de booléens.

Inconvénients :

- si U est très grand, alors t est de grande taille
- si |K| << |U| alors t est creux

4.2.2 Tables de hachage

Soit U l'univers des clés possibles et un tableau t[0..m-1] tel que m < |U|.

Considérons une fonction h : U → {0, 1, …, m-1} qui permette de projeter l'univers des clés sur l'intervalle des indices. h est appelée fonction de hachage. h(k) est appelé valeur de hachage de la clé k.

Un élément de clé k peut être stocké dans l'alvéole d'indice h(k). L'élément de clé k est alors dit haché dans l'alvéole h(k).

Avantages :

- permet de représenter des dictionnaires possédant un grand univers de clés dans un tableau de faible taille
- si h(k) se calcule en O(1), permet de conserver l'efficacité des tables à accès direct

Inconvénient :

- 2 clés distinctes peuvent être hachées dans la même alvéole et provoquer ainsi une collision.

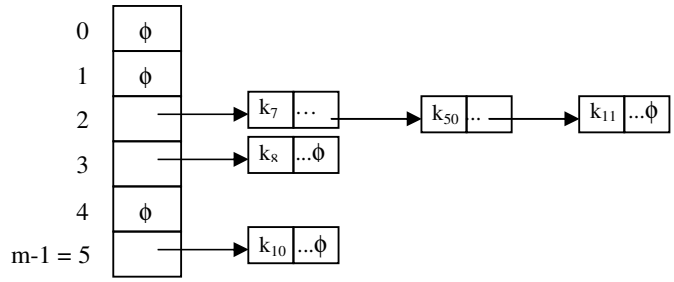
Deux techniques pour résoudre les collisions :

1) le chaînage
2) l'adressage ouvert

4.2.2.1 Résolution des collisions par chaînage

Exemple

En supposant $K = \{k_7 , k_8 , k_{10} , k_{11} , k_{50}\}$, que $h(k_7)=h(k_{11})=h(k_{50})=2$, $h(k_8)=3$ et $h(k_{10})=5$, la table de hachage pourrait être :



Supposons que la table de hachage contienne m alvéoles et mémorise n éléments (|K| = n) et que le calcul de h(k) se réalise en θ(1).

On appelle **facteur de remplissage** le nombre moyen d'éléments stockés dans chaque alvéole. Le facteur de remplissage est défini par $\alpha = n / m$.

La recherche d'un élément prend un temps en θ(1+n) en pire cas (cas où les n éléments sont tous hachés dans la même alvéole). Mais, sous **hypothèse d'un hachage uniforme** – c'est-à-dire que chaque clé a la même probabilité d'être hachée dans l'une quelconque des alvéoles indépendamment de la valeur de hachage des autres éléments – le temps moyen de recherche est en $O(1+\alpha)$. Si n=O(m), alors α est en O(1) et les opérations du dictionnaire peuvent être supportées en un temps en O(1) en moyenne.

Avantages :

- permet de représenter un dictionnaire de capacité non borné
- chacune des opérations de base supportées par le dictionnaire est, en moyenne, très efficace (en O(1) en moyenne)

Inconvénients :

- en pire cas, les fonctionnalités d'exploitation ne sont pas plus efficaces que pour une simple liste chaînée
- l'efficacité repose sur l'hypothèse du hachage uniforme, or rien ne garantit que certains ensembles de clés ne puissent pas provoquer systématiquement un comportement en pire cas (un « ennemi » peut toujours choisir n clés qui seront hachées dans la même alvéole)

4.2.2.2 Fonctions de hachage

Une bonne fonction de hachage doit être rapide à calculer et vérifier au mieux l'**hypothèse du hachage uniforme** : chaque clé a autant de chances d'être hachée dans l'une quelconque des m alvéoles.

La plupart des fonctions de hachage supposent que U est l'ensemble N des entiers naturels. C'est ce que nous supposerons ici. Si toutefois les clés ne sont pas des entiers naturels, un transcodage préalable en entier est donc nécessaire. Par exemple si la clé est une chaîne de caractères, elle peut être interprétée comme un entier exprimé dans une base B adaptée : la clé "$c_0c_1\ldots c_r$" pourrait être transcrite par l'entier :

$$\sum_{i=0}^{r} c_i \, B^{r-i}$$

Deux grandes techniques pour définir un hachage :

1) le hachage par division $h(k) = k \bmod m$
2) le hachage par multiplication $h(k) = \lfloor m \, (k.A \bmod 1) \rfloor$ avec $0<A<1$

où : $\lfloor\ \rfloor$ représente la partie entière
(k.A mod 1) = partie fractionnaire de kA, c'est-à-dire kA - $\lfloor$kA$\rfloor$

Hachage par division.

Intérêt : rapidité de calcul de la valeur de hachage.

Comment choisir m, la taille du tableau t ? Pour tendre vers l'hypothèse du hachage uniforme, on cherchera à utiliser tous les chiffres et tous les bits de k. En pratique, on choisira m tel que :

- m ne soit pas une puissance entière de 2
- m ne soit pas une puissance entière de 10
- m soit premier et éloigné d'une puissance entière de 2

Exemple

Soit n = 2000. Pour 3 éléments en moyenne par alvéole, on choisira une valeur de m proche de 2000/3 et vérifiant les critères ci-dessus. Par exemple : m = 701.

Hachage par multiplication.

Intérêt : la valeur de m n'est pas prise en compte dans le calcul.

Comment choisir m ? On choisit généralement une puissance entière de 2 afin de faciliter le calcul machine.

Comment choisir A ? Un nombre irrationnel est souvent choisi. Knuth[1] suggère le nombre d'or :

$A = (\sqrt{5} - 1) / 2$

4.2.2.3 Résolution des collisions par adressage ouvert

Afin d'éviter la mise en œuvre de pointeurs, tous les éléments sont maintenant conservés dans le table de hachage elle-même. Le facteur de remplissage est donc au plus égal à 1 ($\alpha = n / m \leq 1$).

Intérêt : la mémoire libérée par le non stockage des pointeurs permet d'augmenter le nombre d'alvéoles de la table pour une même quantité mémoire, ce qui engendra moins de collisions et donc des recherches plus rapides.

Au lieu de suivre des pointeurs, on explore une séquence d'alvéoles. Pour déterminer les alvéoles à sonder, on étend la fonction de hachage : un deuxième argument spécifie le numéro du sondage

$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$

Quelle que soit la clé k, la séquence de sondage $< h(k,0), h(k,1), \ldots , h(k,m-1) >$ doit être une permutation de $<0, 1, \ldots, m-1>$.

La procédure d'insertion d'un élément dans la table est donnée ci-après :

[1] Donald E. KNUTH (1938-) est professeur émérite de l'université de Standford. Informaticien renommé, il est un des pionniers de l'algorithmique. Il est à l'origine de nombreux concepts en programmation, d'algorithmes, de logiciels (l'éditeur de texte TEX par exemple), et de la « bible » des informaticiens : « The art of computer programming ».

```
// Insertion d'un élément x de clé k dans une table de hachage gérée
// par adressage ouvert.
Procédure insérer( table de hachage t[0..m-1], Elément x)
      i ← 0 ;
      k ← clé(x) ;
      répéter
            j ← h(k,i) ;           // Hachage de la clé.
            si t[j] = NIL          // Cas alvéole vide.
                  alors t[j] ← x ; // Insérer.
                        retourner ;
                  sinon i ← i+1 ;  // Poursuivre la recherche.
      jusqu'à ce que i = m ;
      erreur « débordement » ;     // Table pleine.
finProcédure
```

L'algorithme de recherche est similaire, la recherche échouant lorsqu'on tombe sur une alvéole NIL.

La suppression d'un élément dans une table à adressage ouvert est plus difficile. Une solution consiste à marquer l'alvéole par un marqueur spécifique VIDEE. L'algorithme *insérer* sera modifié pour traiter ce marqueur comme *NIL* ; la procédure *rechercher* traitera ce marqueur comme *non NIL*. Cette façon de faire présente l'inconvénient de rendre le temps de recherche indépendant du facteur de remplissage α. En pratique, on préférera une technique par chaînage pour résoudre les collisions lorsque des éléments doivent être supprimés.

Techniques de sondage.

L'hypothèse du hachage uniforme devient : chacune des m! permutations possibles de {0, 1, …, m-1} a autant de chance de constituer une séquence de sondage de chaque clé.

Une mauvaise fonction de hachage est une fonction atteinte de la maladie de la grappe : le hachage n'étant pas suffisamment uniforme, des grappes (c.-à-d. des longues suites) d'alvéoles occupées se créent, augmentant ainsi le temps de recherche moyen.

Double hachage.

Le double hachage est une technique performante et très proche du schéma idéal du hachage uniforme :

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

où h_1 et h_2 sont deux fonctions de hachage auxiliaires.

$h_2(k)$ doit être premier par rapport à la taille m du tableau t (c.-à-d. $pgcd(h_2(k),m) = 1$) pour que la table soit parcourue entièrement. Une façon de faire consiste à prendre pour m une puissance entière de 2 et à choisir h_2 telle que : $\forall k \;\; h_2(k)$ impair ; ou bien prendre m premier et h_2 telle que : $\forall k \;\; 0<h_2(k)<m$.

Exemple

m premier
$h_1(k) = k \bmod m$
$h_2(k) = 1 + (k \bmod m')$ ou m' est légèrement inférieur à m (par exemple m-1)
Par exemple : m=701, m'=700.

Efficacité de l'adressage ouvert.

On montre que le nombre moyen de sondage dans le cas $\alpha < 1$ est :

- pour une recherche infructueuse ou une insertion réussie (c.-à-d. pour trouver une alvéole libre) : $< 1/(1-\alpha)$
- pour une recherche fructueuse : $< (1/\alpha) \ln(1/(1-\alpha))$

Exemple

Quel que soit n, si la table est pleine à 90%, une recherche infructueuse nécessitera moins de 10 sondages en moyenne ; une recherche fructueuse nécessitera moins de 2,5 sondages en moyenne.

4.3 Arbres

4.3.1 Arbre vs arborescence

Un **arbre** est un ensemble d'éléments appelés **nœuds,** connectés par liens, et formant un graphe non orienté acyclique connexe.

Une **arborescence** est un arbre « enraciné », c'est-à-dire possédant une racine. L'existence d'une racine impose un sens de parcours et donc des liens orientés, appelés **arcs**. Une arborescence est donc un ensemble de nœuds organisés en une hiérarchie descendante à partir du nœud racine.

Les arborescences sont des structures de données parmi les plus importantes et spécifiques de l'informatique. *Exemples d'application : organisation d'un système de fichiers, représentation syntaxique d'un programme source par un compilateur, vérificateur d'orthographe, dictionnaire des éléments de gestion d'une base de données, file d'attente avec gestion de priorités, …*

Dans une arborescence :

- Un nœud y est dit **fils** du noeud x ssi il existe un arc de x à y.
- Le **degré** d'un noeud x est le nombre de fils de ce noeud.
- Une **arborescence binaire** est une arborescence dont tout nœud a un degré inférieur ou égal à 2.
- Une **feuille** (ou nœud externe) est un nœud de degré 0 ; un **nœud interne** est un nœud de degré non nul.
- Un **chemin** menant de x_i à x_j est une suite de nœuds consécutifs $< x_i, x_{i+1}, \dots x_j >$ tels que $\forall\ x_k \in]i, j]$ x_k est fils de x_{k-1}. Propriété : si un chemin existe entre deux nœuds, il est unique.
- Tout nœud sur le chemin menant de la racine à ce nœud est appelé **ancêtre** de ce nœud ; x est ancêtre de y $\Leftrightarrow$ y est **descendant** de x.
- La **profondeur d'un nœud** est la longueur, en nombre d'arcs, du chemin de la racine à ce noeud.
- La **hauteur d'un nœud** est la longueur, en nombre d'arcs, du plus long chemin reliant ce noeud à une feuille.
- La **hauteur d'une arborescence** est la profondeur du nœud le plus profond (c'est-à-dire la hauteur de la racine).

<u>En informatique, tous les arbres utilisés étant en général des arborescences, le terme arbre est généralement utilisé en lieu et place du terme arborescence. C'est cette convention qui sera utilisée par la suite</u>.

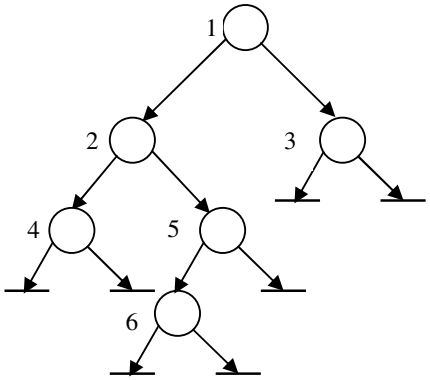
4.3.2 Arbre binaire

4.3.2.1 Définition et propriétés

Un **arbre binaire** (en toute rigueur une arborescence binaire) ab est une arborescence qui est :

- soit vide, c'est-à-dire ne contient aucun nœud. On notera ab = NIL.
- soit définie par un triplet de la forme ab = <r, abg, abd>
  où r, abd, abg constituent trois ensembles de nœuds disjoints : r est le noeud **racine**, abg est un arbre binaire appelé **sous-arbre gauche**, et abd est un arbre binaire appelé **sous-arbre droit**

Exemple, les nœuds étant numérotés dans l'ordre naturel (de haut en bas, de gauche à droite) :



Définitions et propriétés :

- Un nœud unique est lui-même un arbre : <r, NIL, NIL>
- Un arbre binaire quelconque n'a pas de symétrie gauche-droite : <r, a, b> ≠ <r, b, a>
- La racine du sous-arbre gauche d'un nœud x, si ce sous-arbre existe, est appelé **fils gauche** de x ; la racine du sous-arbre droit de x, si ce sous-arbre existe, est appelé **fils droit** de x. Tout nœud a 0 ou 1 fils gauche et 0 ou 1 fils droit.
- La hauteur d'un arbre binaire non vide de n nœuds vérifie : $\lceil \log_2(n+1) \rceil - 1 \leq h \leq n-1$
- La hauteur d'un arbre binaire non vide de f feuilles vérifie : $h \geq \lceil \log_2(f) \rceil$
- Le nombre f de feuilles d'un arbre binaire de n nœuds vérifie : $f \leq (n+1)/2$

4.3.2.2 Implémentation

Typiquement, les nœuds sont implémentés par des structures dynamiques chaînées par des liens père → fils. Chaque nœud contient : une clé et ses données satellites, un pointeur vers le sous-arbre gauche et un pointeur vers le sous-arbre droit. Un arbre binaire ab est un pointeur vers le nœud racine. Si ab est un arbre non vide, on conviendra de noter :

```
clé[ab]     la variable clé du nœud racine de ab
gauche[ab]  la variable pointeur vers le sous-arbre gauche de ab
droite[ab]  la varaible pointeur vers le sous-arbre droit de ab
```

4.3.2.3 Topologies particulières

Topologies particulières d'arbres binaires :

- Un **arbre binaire dégénéré** (ou **arbre binaire filiforme**) est un arbre binaire dont tous les nœuds internes sont de degré 1. Un arbre binaire filiforme de hauteur h possède h+1 nœuds.
- Un **arbre binaire complet**[1] est un arbre binaire dont tous les nœuds internes sont de degré 2.
- Un **arbre binaire parfait**[2] est un arbre binaire complet dont toutes les feuilles sont à la même profondeur. Un arbre binaire parfait de hauteur h possède 2^{h+1}-1 nœuds.
- Un **arbre binaire presque parfait**[3] de hauteur h est un arbre binaire parfait jusqu'à la profondeur h-1, et dont toutes les feuilles de profondeur h sont « le plus à gauche possible ».

  Propriété. Tout arbre binaire presque parfait de n nœuds peut être représenté par un tableau t[0..n-1] tel que :

  t[0] = nœud racine

  $\forall i \in [1..n-1]$ si t[i] = nœud i
  alors $2i+1<n \Rightarrow$ t[2i+1] = fils gauche de i
  $2i+2<n \Rightarrow$ t[2i+2] = fils droit de i

  et réciproquement : tout tableau t[0..n-1] peut être vu comme la représentation d'un arbre binaire presque parfait de n nœuds.
- Un **arbre binaire équilibré** est un arbre binaire tel que, en tout nœud, la différence de hauteur des sous-arbres gauche et droit diffère au plus de 1. La hauteur h d'un arbre binaire équilibré de n nœuds vérifie : $\log_2(n+1) \leq h \leq 1{,}44 \log_2(n+2)$

4.3.2.4 Algorithmes de parcours

A la base de tous les algorithmes sur des arbres binaires quelconques, il y a le **parcours** d'arbre. Parcourir un arbre c'est examiner systématiquement, dans un certain ordre, chacun des nœuds de l'arbre pour y effectuer un traitement donné. Un arbre peut être parcouru en largeur d'abord (c'est-à-dire niveau de profondeur par niveau de profondeur) ou en profondeur d'abord (c'est-à-dire par branche). Dans les deux cas, le parcours systématique de tous les nœuds nécessite de mettre en œuvre une liste d'attente (pile ou file) de noeuds.

Parcours en profondeur.

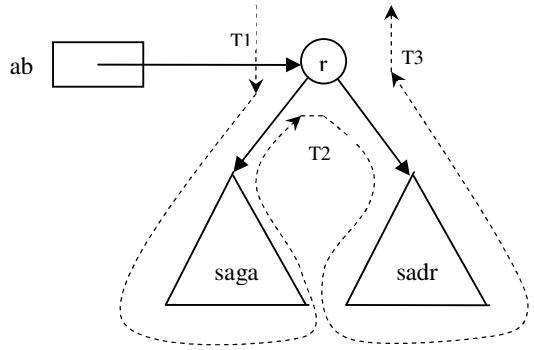
Considérons un parcours en profondeur, plus précisément un parcours en profondeur à main gauche. Un tel parcours consiste à tourner autour de l'arbre en partant de la racine, en le serrant au plus près, sans le couper, et en le laissant à main gauche, comme l'illustre le schéma ci-après sur un arbre binaire ab non vide <r, saga, sadr>.

Dans ce parcours, chaque sous-arbre vide est rencontré 1 fois (notons T0 le traitement qu'on réalise alors), et chaque nœud est rencontré 3 fois : la première à la descente dans l'arbre (notons T1 le traitement qu'on réalise alors), la deuxième lors de la remontée intermédiaire (notons T2 le traitement qu'on réalise alors), et la troisième lors de la remontée finale (notons T3 le traitement qu'on réalise alors).

[1] Définition non stable dans la littérature informatique.

[2] Idem.

[3] Idem.




```
// Parcours en profondeur à main gauche d'un arbre binaire ab.
Procédure parcours (ArbreBinaire ab)
      si ab = NIL alors T0 ;
      sinon T1 ;                    // A la descente
            parcours(gauche[ab]) ;
            T2 ;                    // Après gauche et avant droite
            parcours(droite[ab]) ;
            T3 ;                    // A la remontée
      finSi ;
finProcédure
```

La pile gérée par la récursivité de cette procédure mémorise le chemin de descente dans l'arbre au fil des appels récursifs et permet ainsi de remonter dans l'arbre au fil des retours d'appels récursifs afin de pouvoir explorer ensuite d'autres branches. La mise en œuvre d'une pile – implicite via la récursivité ou explicite par une variable de type Pile – est la seule façon de faire pour remonter dans l'arbre quand on ne dispose pas de liens fils → père.

Cas particuliers classiques de parcours :

- **parcours préfixe** : traitement des nœuds à la 1ère rencontre → que T1 (et T0)
- **parcours infixe** : traitement des nœuds à la 2ème rencontre → que T2 (et T0)
- **parcours postfixe** : traitement des nœuds à la 3ème rencontre → que T3 (et T0)

Exemple. Ordre de traitement des nœuds de l'arbre binaire de l'exemple précédent :

- *dans un parcours préfixe : 1, 2, 4, 5, 6, 3*
- *dans un parcours infixe : 4, 2, 6, 5, 1, 3*
- *dans un parcours postfixe : 4, 6, 5, 2, 3, 1*

Exemple. La recherche d'un élément dans un arbre binaire quelconque de n nœuds se réalise par une fonction de parcours préfixe en O(n).

```
// Recherche d'un élément de clé k dans un arbre binaire ab
// quelconque. La fonction renvoie NIL ou un pointeur sur l'élément.
Fonction recherche(ArbreBinaire ab, Clé k) : ArbreBinaire
      si (ab = NIL) ∨ (clé[ab] = k) alors retourner ab ;
      p ← recherche(gauche[ab], k) ;
      si p ≠ NIL alors retourner p ; finSi ;
      retourner recherche(droite[ab], k) ; ;
finFonction
```

Parcours en profondeur itératif.

Un parcours en profondeur peut être implémenté de façon itérative, mais en mettant explicitement en œuvre une pile.

```
// Parcours en profondeur d'un arbre binaire ab quelconque.
// Version préfixe itérative.

Procédure parcoursEnProfondeur(ArbreBinaire ab)
    si ab = NIL alors T0 ;
    sinon
       p ← pile vide ;
       empiler(p, ab) ;      // empile l'adresse de racine(ab).
       tant que p n'est pas vide faire
          ab ← dépiler(p) ; // extrait le dernier élément empilé.
          T1 ;              // traite la racine de ab.
          si droite[ab] ≠ NIL alors empiler(p, droite [ab]) ; finSi ;
          si gauche[ab] ≠ NIL alors empiler(p, gauche[ab]) ; finSi ;
      finTantQue ;
    finSi ;
finProcédure
```

Parcours en largeur.

Un parcours en largeur consiste à explorer d'abord les nœuds de même profondeur avant de poursuivre avec les nœuds du niveau de profondeur suivant. L'algorithme, itératif, met en œuvre une file d'attente (voir exemple section 9.1.1).

```
// Parcours en largeur d'un arbre binaire ab quelconque.
// Version préfixe itérative.

Procédure parcoursEnLargeur(ArbreBinaire ab)
    si ab = NIL alors T0 ;
    sinon
       f ← file d'attente vide ;
       ajouter(f, ab) ;      // met en file l'adresse de racine(ab).
       tant que f n'est pas vide faire
          ab ← retirer(f) ; // extrait le 1er élément de la file.
          T1 ;              // traite la racine de ab.
          si gauche[ab] ≠ NIL alors ajouter(f, gauche[ab]) ; finSi ;
          si droite[ab] ≠ NIL alors ajouter(f, droite[ab]) ; finSi ;
      finTantQue ;
    finSi ;
finProcédure
```

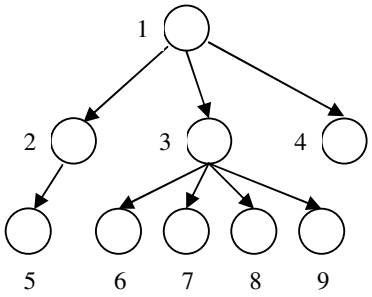
Exemple. Ordre de traitement des nœuds de l'arbre binaire de l'exemple précédent :1, 2, 3, 4, 5, 6.

4.3.3 Arbre général

Un arbre général (en toute rigueur une arborescence générale) est une arborescence dans laquelle tout nœud peut posséder un nombre quelconque de fils : chaque nœud x a un degré $d(x) \geq 0$.

Dans un arbre général, il n'y a plus de notion de fils gauche ou fils droit, mais de 1er, 2ème, …, kième fils, dans une lecture gauche-doite. Le 1er fils, situé le plus à gauche, est appelé **fils aîné**. On appelle **frère droit** d'un nœud x le frère de x situé immédiatement à sa droite.

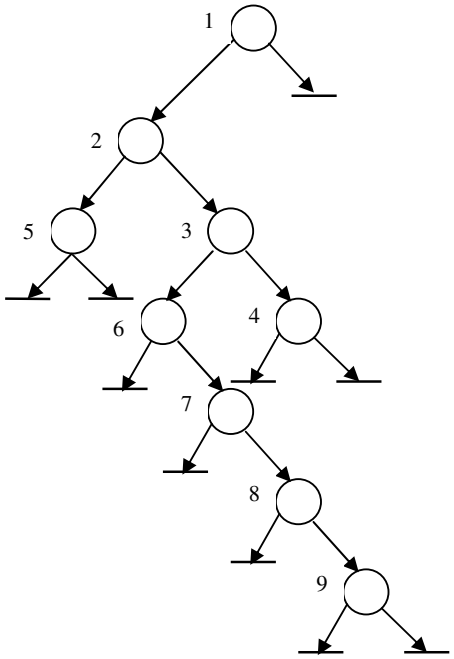
Exemple :



Une représentation usuelle d'un arbre général est la représentation par l'arbre binaire « fils aîné – frère droit » : le lien gauche relie au fils aîné, le lien droit relie au frère droit.

Exemple

L'arbre général précédent pourrait se représenter par l'arbre binaire suivant :



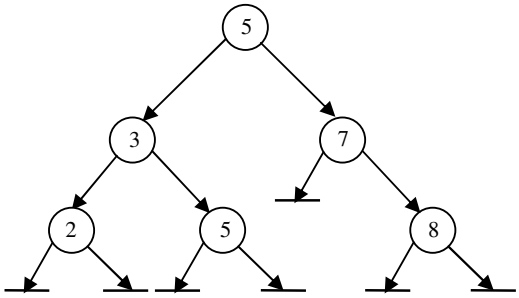
4.3.4 Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire possédant la propriété suivante :

Quel que soit le nœud n :

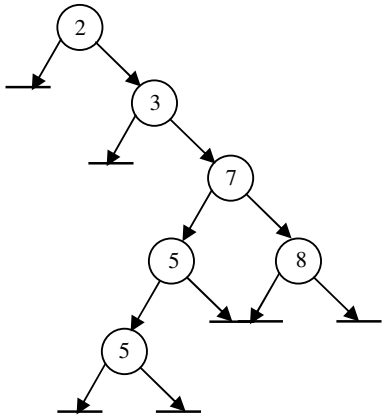
- tous les nœuds du sous-arbre gauche de n, s'il existe, ont une clé inférieure ou égale à celle de n
- tous les nœuds du sous-arbre droit de n, s'il existe, ont une clé strictement supérieure à celle de n

Exemple, chaque nœud étant étiqueté par sa clé :



Deux arbres binaires de recherche différents peuvent représenter le même ensemble de valeurs.

Exemple. L'arbre binaire de recherche suivant représente le même ensemble de clés que le précédent



Propriétés des arbres binaires de recherche :

- les clés sont ordonnées horizontalement par valeurs croissantes
- un parcours infixe traite la suite des clés par ordre croissant de valeur

L'intérêt des arbres binaires de recherche réside dans l'efficacité des opérations de base sur les ensembles dynamiques : les opérations rechercher, insérer, supprimer, minimum, maximum, prédécesseur et successeur peuvent en effet s'exécuter de façon itérative en O(h), où h est la hauteur de l'arbre.

```
// Recherche d'un élément de clé k dans un arbre binaire de recherche
// ab. La fonction renvoie NIL ou un pointeur sur l'élément.
Fonction recherche(ABR ab, Clé k) : ABR
      p ← ab ;
      tant que (p ≠ NIL)  ∧ (clé[p] ≠ k) faire
            si k ≤ clé[p]
                alors p ← gauche[p] ;
                sinon p ← droite[p] ;
            finSi ;
      finTantQue ;
      retourner p ;
finFonction
```

```
// Insertion d'un élément x de clé k dans un arbre binaire de
// recherche ab.
// Opère toujours par insertion d'une feuille x.

Procédure insérer(ABR ab, Elément x)
      p ← ab ;
      pp ← NIL ;  // père de p
      k ← clé[x] ;
      tant que p ≠ NIL faire
            pp ← p ;
            si k ≤ clé[pp]
                alors p ← gauche[p] ;
                sinon p ← droite[p] ;
            finSi ;
      finTantQue ;
      p ← <x, NIL, NIL>
      si pp = NIL alors ab ← p ;                  // ab initial vide.
      sinon si k ≤ clé[pp] alors gauche[pp] ← p ; // Insertion à gche
      sinon droite[pp] ← p ;                      // Insertion à dte
finProcédure
```

Autres opérations (pour être efficaces, ces opérations nécessitent de mémoriser dans chaque nœud un lien vers le nœud père) :

- Déterminer le successeur* d'un nœud. Le successeur d'un nœud x est, s'il existe, le nœud y dont la clé est la plus petite des clés plus grandes que clé[x]. Si le sous-arbre droit de x n'est pas vide, alors le successeur y est le nœud « le plus à gauche » du sous-arbre droit de x ; si le sous-arbre droit de x est vide, alors le successeur de x est le premier ancêtre de x dont le fils gauche est x ou ancêtre de x.
- Supprimer un nœud. Si le nœud x à supprimer n'a pas de fils, alors il suffit de le supprimer ; si le nœud x à supprimer n'a qu'un seul fils, on détache x en créant un nouveau lien entre le père de x et le fils de x ; si le nœud x à supprimer a deux fils*, on détache le nœud y qui est le plus grand des plus petits descendants de x (le prédécesseur* de x) et on remplace la clé et les éventuelles données satellites de x par celles de y.

*Nota. Propriété : si un nœud a deux fils, alors son successeur n'a pas de fils gauche et son prédécesseur n'a pas de fils droit.

4.3.5 Arbre rouge-noir

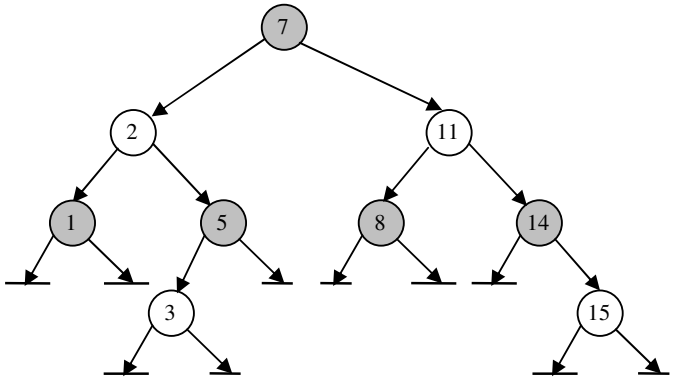
L'efficacité des opérations sur les arbres binaires de recherche est conditionnée au caractère raisonnablement équilibré de l'arbre. Les arbres rouge-noir sont des arbres binaires de recherche qui sont approximativement équilibrés au sens où aucun chemin de la racine à un nœud quelconque n'est plus de deux fois plus long que n'importe quel autre. Pour maintenir cette propriété, un bit supplémentaire est stocké dans chaque nœud : il représente la couleur du nœud, rouge ou noir.

Un arbre binaire de recherche est un arbre rouge-noir s'il satisfait aux propriétés suivantes :

- chaque nœud est soit rouge soit noir
- chaque sous-arbre vide représente un nœud sentinelle noir
- si un nœud est rouge, ses fils sont noirs
- pour chaque nœud, tous les chemins reliant le nœud à une feuille d'un de ses sous-arbres contiennent le même nombre de nœuds noirs.

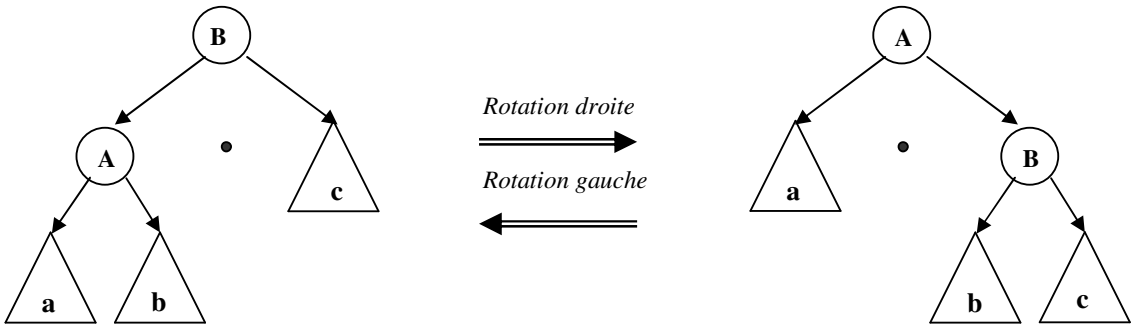
La seconde condition est une définition purement technique. La troisième condition stipule que les nœuds rouges ne sont pas trop nombreux. La dernière condition est une condition d'équilibre : elle signifie que si l'on oublie les nœuds rouges on obtient un arbre parfaitement équilibré.

Exemple d'arbre rouge-noir (nota : l'alternance noir → rouge est spécifique à cet exemple et n'est pas de règle) :



La hauteur h d'un arbre rouge-noir de n nœuds vérifie : $h \leq 2 \log_2(n+1)$.

Quand on insère ou on supprime un nœud dans un arbre binaire de recherche « équilibré », on peut rompre la propriété d'équilibre. L'opération de base pour restaurer cette propriété – tout en préservant la propriété d'arbre binaire de recherche ! – est la rotation.



Les dénominations rotation droite vs gauche sont à considérer au sens d'une rotation des nœuds A et B autour du point indiqué.

Dans un arbre rouge-noir, le rééquilibrage après l'insertion ou la suppression d'un nœud ne requiert jamais plus de 3 rotations.

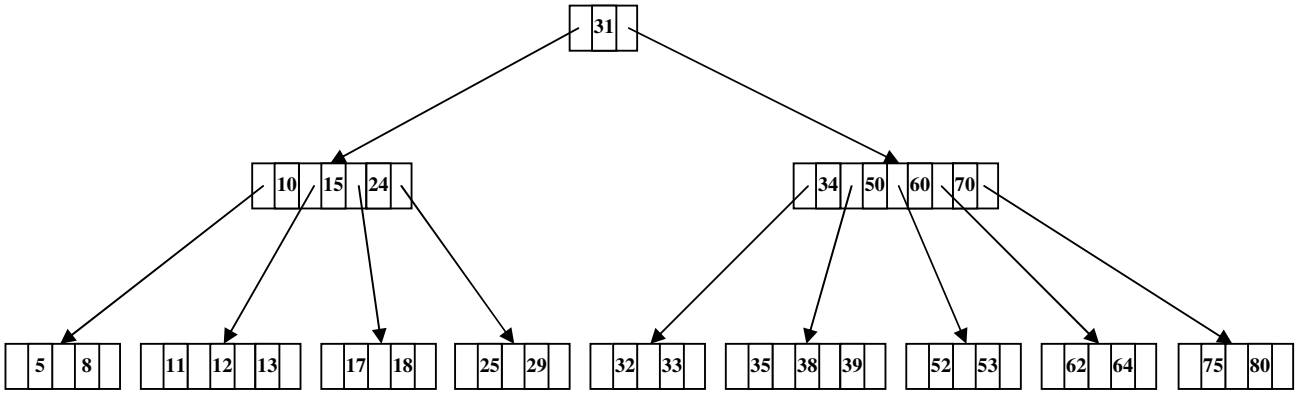
4.3.6 Arbre-B (B-Tree)

Les arbres-B (ou B-Arbres) sont des arbres de recherche équilibrés – par construction – utilisés dans l'implémentation de bases de données et de systèmes de fichiers. Ils permettent aux nœuds de posséder plus d'une clé, ce qui réduit la taille de l'arbre et le nombre d'opérations de rééquilibrage. Chaque nœud interne n possède k_n+1 fils, où k_n est le nombre de clés de ce noeud (en pratique, k peut être très grand, jusqu'à quelques milliers !). La recherche dans un arbre B s'effectue comme dans un arbre binaire de recherche. Toutefois, contrairement aux arbres binaires de recherche, ce sont des arbres qui grandissent par la racine et non par les feuilles.

Un arbre-B d'ordre (ou degré minimal) m est tel que :

- chaque nœud contient k clés triées, avec $m \leq k \leq 2m$ sauf la racine pour laquelle k vérifie $1 \leq k \leq 2m$.
- tout nœud interne a (k+1) fils ; le $i^{ème}$ fils a des clés comprises entre les $(i-1)^{ème}$ et $i^{ème}$ clés du père.
- l'arbre est équilibré.

Exemple d'arbre-B d'ordre 2 :



La hauteur h d'un arbre-B d'ordre m de n nœuds, n≥1, vérifie : $h \leq \log_m(\frac{n+1}{2})$

Principe d'organisation des fichiers sur disque et de mise en œuvre d'un arbre B. Un disque magnétique est partitionné en pages (par exemple de taille 2048, 4092 ou 8192 octets). La page est l'unité de transfert entre mémoire centrale et disque. Le temps de positionnement de la tête de lecture sur une page disque étant environ 10^5 fois plus lente que l'accès direct à une information en mémoire centrale, l'efficacité impose donc de minimiser le nombre d'accès disque. C'est ce qu'apporte un arbre-B. Dans un tel arbre, les clés des données sont regroupées par nœud, chaque nœud ayant typiquement la taille d'une page disque et chaque nœud requérant un seul accès disque. La hauteur de l'arbre mesure ainsi le nombre d'accès disque nécessaire. Et cette hauteur est très faible. Par exemple, si chaque nœud contient environ 1000 clés, il suffit d'un arbre de hauteur 3 pour stocker un milliard de clés.

4.3.7 Arbre binaire de recherche optimal

Soit un ensemble de n clés distinctes $C = \{c_0, c_1, \ldots, c_n\}$. Pour chaque clé c_i, on suppose connaître la probabilité p_i qu'une recherche concerne cette clé. On suppose aussi possible des recherches de clés n'appartenant pas à C. Comment construire un arbre binaire de recherche qui minimise le coût de recherche moyen d'une clé ?

Ce problème relève de la programmation dynamique (dont le principe est introduit au chapitre 7). La solution, en $\theta(n^3)$, ne sera pas développée ici. On peut toutefois souligner que :

- un arbre binaire de recherche optimal n'est pas nécessairement un arbre de hauteur minimal ;
- la clé racine d'un arbre binaire de recherche optimal n'est pas nécessairement celle de plus grande probabilité.

5 ALGORITHMES DE TRI

Trier, c'est organiser selon une certaine relation d'ordre.

On distingue habituellement les tris internes (l'ensemble des données à trier est stocké en mémoire centrale) des tris externes (l'ensemble des données à trier est stocké sur disque). Les algorithmes de tri externe sont moins fondamentaux maintenant que les données pertinentes sont le plus souvent mises en mémoire virtuelle. Les tris externes sont hors du champ couvert ici.

Par la suite, nous nous limiterons à des algorithmes de **tri interne**.

Outre sa complexité algorithmique (en moyenne vs en pire cas) et son besoin d'espace mémoire additionnel, un tri peut avoir des caractéristiques particulières :

- **tri sur place** : un algorithme trie sur place s'il n'utilise qu'un nombre très limité de variables et modifie directement la structure qu'il est en train de trier sans devoir faire usage à une structure de données tampon.
- **tri stable** : un algorithme de tri est dit stable s'il garde l'ordre relatif des éléments égaux pour la relation d'ordre. Cette propriété permet de trier successivement un tableau selon plusieurs clés et de conserver l'ordre obtenu lors des tris précédents.

Par la suite, on se pose le problème de trier par ordre croissant les éléments d'un tableau t[0..n-1] – voire d'une liste – c'est-à-dire de trouver un programme P répondant à l'énoncé :

$$(t[0..n-1]=[e_0, e_1, \ldots, e_{n-1}] \wedge n>0) \{ P \} (t[0..n-1]=\prod(e_0, e_1, \ldots, e_{n-1}) \wedge t[0..n-1]\uparrow)$$

où $\prod(e_0, e_1, \ldots, e_{n-1})$ est une permutation des éléments de $[e_0, e_1, \ldots, e_{n-1}]$ et $t[0..n-1]\uparrow$ dénote un tableau t[0..n-1] trié par ordre croissant d'éléments.

5.1 Tris par comparaisons

5.1.1 Tri par sélection

Le tri par sélection est un des nombreux algorithmes de tri « naïfs » mais peu efficaces car en $\theta(n^2)$.

Invariant possible : $I(i) = (t[i..n-1]\uparrow \wedge t[0..i-1] \leq t[i])$

Condition d'arrêt : i=1 car alors $t[0..n-1]\uparrow$

Corps de boucle :
```
// I(i) ∧ i≠1
monterMax(t[0..i-1], i-1) ;
// I(i-1)
i ← i-1 ;
// I(i)
```

où monterMax(t[0..i-1], i-1) est une procédure qui recherche le plus grand élément de t[0..i-1] et le permute avec t[i-1].

Initialisations : `i ← n ; // I(i)`

La justification de la propriété I(i) à l'issue des initialisations repose sur la sémantique de l'invariant, qui s'énonce en fait :

I(i) = ($1 \le i \le n-1 \Rightarrow (t[i..n-1]\uparrow \wedge t[0..i-1] \le t[i])$)

où $\Rightarrow$ dénote l'implication logique, et donc I(i) est vrai si i=n.

Reste à écrire une procédure monterMax(t[0..i-1], i-1) répondant à l'énoncé :

$$(t[0..i-1]=[t_0, t_1, \ldots, t_{n-1}] \wedge i>0)$$
$$\{ \text{monterMax}(t[0..i-1], i-1) \}$$
$$(t[0..i-1]=\prod(t_0, t_1, \ldots, t_{i-1}) \wedge t[i-1]=\max(t[0..i-1]))$$

La procédure monterMax(t[0..i-1], i-1) consiste à :

1) chercher l'indice istar du max(t[0..i-1]) ;

2) permuter t[i-1] et t[istar].

Recherchons istar :

Invariant possible : I'(istar, j) = (t[istar]= max(t[0..j-1]))

Condition d'arrêt : j=i

Corps de boucle :

1^{er} cas : $I'(istar, j) \wedge t[j]>t[istar] \Rightarrow I(j, j+1)$

$2^{ème}$ cas : $I'(istar, j) \wedge t[j] \le t[istar] \Rightarrow I(istar, j+1)$

Initialisations : j ← 1 ; istar ← 0 ; // I(istar, j)

L'algorithme de tri par sélection s'écrit donc :

```
// « Tri par sélection » de t[0..n-1], avec n>0

Pour i variant de n à 2 par pas de -1 faire
      // I(i) = ( t[i..n-1]↑ ∧ t[0..i-1]≤t[i] )

      istar ← 0 ;
      Pour j variant de 1 à i-1 par pas de 1 faire
            //I'(istar, j) = ( t[istar]= max(t[0..j-1]) )
            si t[j]>t[istar] alors
                  // I'(j, j+1)
                  istar ← j ;
                  // I'(istar, j+1)
            finSi ;
            // I'(istar, j+1)
      finPour ;
      // I'(istar, i)
      échanger t[istar] et t[i-1]
      // I(i-1)

finPour ;
// t[0..n-1]↑
```

Quelle est la complexité de cet algorithme ? Chaque opération élémentaire étant en $\theta(1)$, le temps d'exécution de monterMax(t[0..i-1], i-1) est de la forme a+b i, où a et b sont des constantes. Le temps de calcul total du tri par sélection est donc de la forme :

$$T(\text{triParSélection}, t[0..n-1]) = c + \sum_{i=2}^{n}(a+b.i) \qquad \text{où c est une constante}$$
$$= \alpha + \beta n + \gamma n^2 \qquad \text{où } \alpha, \beta \text{ et } \gamma \text{ sont des constantes}$$
$$= \theta(n^2)$$

Le tri par sélection est donc de complexité quadratique.

5.1.2 Tri par insertion

Le tri par insertion est un autre algorithme de tri « naïf » en $\theta(n^2)$. Par rapport au tri par sélection, au tri bulle, …, il présente néanmoins quelques avantages :

- il peut être utilisé pour insérer un nouvel élément dans un tableau déjà trié
- il peut être utilisé pour trier des valeurs au fur et à mesure de leur apparition
- sa complexité est en $\theta(n)$ si le tableau initial est déjà trié dans l'ordre souhaité.

Principe. L'algorithme est celui qu'on met en œuvre quand, au jeu de cartes, on classe ses cartes au fur et à mesure qu'on les tire : on tire une nouvelle carte et on l'insère dans la main déjà ordonnée.

L'algorithme de tri par insertion s'écrit :

```
// « Tri par insertion » de t[0..n-1], avec n>0

Pour i variant de 1 à n-1 par pas de 1 faire
      // I(i) = ( t[0..i-1]↑ )
      e ← t[i] ; // « libère » la cellule t[i] ;
                 // e est à classer dans t[0..i-1]
      j ← i-1 ;
      // I'(j) = ( (t[j+1] libre) ∧
      //           (0≤j<i ⇒ t[0..j]↑)  ∧
      //           (-1≤j≤i-2 ⇒ e<t[j+2..i]↑)  ∧
      //           (0≤j≤i-2 ⇒ t[0..j]↑≤t[j+2..i]↑)
      //          )
      tant que j≥0  ∧ e<t[j] faire
            // I'(j) ∧  j≥0  ∧ e<t[j]
            t[j+1] ← t[j] ;
            // (j>0 ⇒ t[0..j-1]↑) ∧ (t[j] libre) ∧ e<t[j+1..i]↑
            j ← j-1 ;
            // I'(j)
      finTantQue ;
      // I'(j) ∧  (j≠-1 ⇒ e≥t[j])
      t[j+1] ← e ;
      // I(i+1)

finPour ;
// t[0..n-1]↑
```

5.1.3 Tri rapide (Quick Sort)

Le célèbre Quick Sort est l'un des algorithmes de tri les plus performants. Il a été proposé par Hoare. Il repose sur le paradigme « Diviser pour régner ».

Le paradigme[1] « **Diviser pour régner** »[2] consiste à :

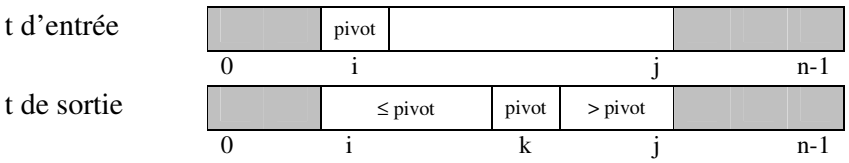
1) diviser le problème en sous-problèmes de la même nature ;
2) régner sur chaque sous-problème en le résolvant (directement quand la taille du sous-problème est suffisamment petite, ou de manière récursive dans le cas général) ;
3) combiner les solutions des sous-problèmes pour produire la solution du problème originel.

5.1.3.1 Algorithme générique

La procédure trier(t[i..j]) peut être réalisée ainsi :

- cas i<j
  partitionner(t, i, j, k) pour que, in fine, t[i..k-1]≤t[k]<t[k+1..j] ; // diviser
  trier(t[i..k-1]) ; // régner
  trier(t[k+1..j] ; // régner
- cas i=j : t[i..j] contient un seul élément et est donc trié
- cas i>j : t[i..j] contient zéro élément et est donc trié

La procédure partitionner(t, i, j, k) se veut réaliser l'opération suivante :



Le pivot est la valeur de l'élément t[i] initial, valeur pivot du partitionnement de t[i..j] ; k est un paramètre de sortie (et non d'entrée !!)

La procédure Quick Sort s'écrit donc :

```
// « Quick Sort générique » - Tri de t[i..j].
// Appel principal pour trier t[0..n-1] : QuickSort (t, 0, n-1).
Procédure QuickSort (t, i, j)
      si i<j alors
            Partitionner(t, i, j, k) ;// t[i..k-1]≤t[k]<t[k+1..j]
            QuickSort (t, i, k-1) ;  // t[i..k-1]↑≤t[k]<t[k+1..j]
            QuickSort (t, k+1, j) ;  // t[i..k-1]↑≤t[k]<t[k+1..j]↑
      finSi ;
      // t[i..j]↑
finProcédure
```

[1] Un paradigme est une façon de voir les choses qui repose sur un modèle.

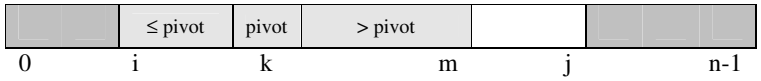
[2] Tout algorithme « diviser pour régner » (*Divide and conquer*) est par nature récursif. La réciproque est fausse (tout algorithme récursif ne réduisant pas la taille de l'exemplaire en cours de traitement n'est pas, par définition, un algorithme de type « diviser pour régner »). En outre, l'appellation « diviser pour régner » ne devrait être employée que quand chaque problème peut se diviser en deux sous-problèmes ou plus. Si chaque problème ne se réduit qu'à un seul sous-problème, on préfèrera l'expression (moins courante) « réduire pour régner » (*Decrease and conquer*).

Il reste à écrire une procédure partitionner(t, i, j ,k) répondant à l'énoncé :

$$(t[i..j]=[e_i, e_{i+1}, \ldots, e_j] \ \wedge \ i<j)$$
$$\{ \text{ partitionner(t, i, j, k) } \}$$
$$(t[i..j]=\prod(e_i, e_{i+1}, \ldots, e_j) \ \wedge \ t[i..k-1] \leq t[k] < t[k+1..j])$$

Invariant possible :

I(k, m) =



I(k, m) = (t[i..k-1]≤t[k]<t[k+1..m])

Condition d'arrêt : m=j car alors la post-condition est vraie

Corps de boucle :

1er cas : I(k, m) ∧ t[m+1]>t[k] ⇒ I(k, m+1)
2ème cas: I(k, m) ∧ t[m+1]≤t[k]
{ permutation circulaire droite de t[k], t[k+1] et t[m+1] }
I(k+1, m+1)

Initialisations : k ← i ;
m ← k ;
// I(k, m)

La justification de la propriété I(k, m) à l'issue des initialisations repose sur la sémantique de l'invariant, qui s'énonce en fait :

I(k, m) = (0≤i<k<m≤j≤n-1 ⇒ (t[i..k-1]≤t[k]<t[k+1..m])

où ⇒ dénote l'implication logique, et donc I(k, m) est vrai si k=i et m=k.

L'algorithme partitionner(t, i, j, k) est écrit ci-après. Il a une complexité en θ(j-i).

```
// Algorithme Partitionner(t[0..n-1], i, j, k), 0≤i<j≤n-1
// Paramètres d'entrée : t, i, j
// Paramètre de sortie : k

// 0≤i<j≤n-1
k ← i ;
m ← i ;
// I(k, m) = ( t[i..k-1]≤t[k]<t[k+1..m]  )
tant que m ≠ j faire
      // I(k, m) ∧ m<j
      m ← m+1 ;
      // I(k, m-1) ∧ m≤j
      si t[m] ≤ t[k] alors
            permutation circulaire droite de t[k], t[k+1] et t[m] ;
            // I(k+1, m) ∧ m≤j
            k ← k+1 ;
            // I(k, m) ∧ m≤j
      finSi ;
      // I(k, m) ∧ m≤j
fintantQue ;
// t[i..k-1]≤t[k]<t[k+1..j]
```

5.1.3.2 Complexité algorithmique

Quel est le temps de calcul T(n) de la procédure Quick Sort appliquée à un tableau de n éléments?

Supposons d'abord que $n=2^p$, avec $p \in N$, et que, à chaque appel, le sous-tableau t[i..j] soit séparé en k = (i+j)/2.

T(n) est défini par la relation de récurrence :

$T(n=2^0) = a$

$T(n=2^p) = a + (b + c.n) + T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) \qquad \forall\, p \in N^*$

où (b + c.n) est le temps de calcul de partitionner(t, 0, n, k), et a, b et c des constantes.

$T(n=2^p)$ peut être borné supérieurement :

$T(n=2^p) \leq a + (b + c.n) + 2T(n/2)$

Le développant de l'équation de récurrence conduit alors à :

$$T(n=2^p) \leq a\,(2^{p+1}-1) + b\sum_{i=0}^{p-1} 2^i + c.p.2^p$$

Donc $T(n=2^p) \leq a\,(2^{p+1}-1) + b\,(2^p-1) + c.p.2^p$

$T(n=2^p) \leq \alpha + \beta\, n + \gamma\, n \log_2(n)$ où α, β et γ sont des constantes

Il s'en suit que T(n) = O(n.log(n))

Considérons maintenant les cas où les hypothèses précédentes ne sont pas satisfaites.

- 1^{er} cas : $n \neq 2^p$, avec $p \in N^*$. Dans ce cas, $2^p < n < 2^{p+1}$. T(n) étant croissante en n, on peut interpoler pour des valeurs qui ne sont pas puissances de 2. L'arbre des appels de la procédure Quick Sort est de hauteur $\theta(\lfloor \log(n) \rfloor)$ et chaque niveau de profondeur a un coût total en O(n), donc que T(n) = O(n.log(n)).
- $2^{ème}$ cas : la division du sous-tableau t[i..j] à chaque étape ne divise pas systématiquement par 2 la plage [i..j].

  Le cas le pire se présente lorsque chaque appel de partitionner(t, i, j, k) produit :

  soit k=i (ce cas se présente quand t[0..n-1]↑ initialement)

  soit k=j (ce cas se présente quand t[0..n-1]↓ initialement)

  Alors T(n) est défini par la récurrence :

  $T(n=0) = T(n=1) = a$

  $T(n>1) = a + (b + c.n) + T(0) + T(n-1)$

  Le développement de la récurrence conduit à :

  $$T(n>1) = a\,(2n - 1) + b\,(n-1) + a\sum_{i=1}^{n} i - a$$

  Donc $T(n>1) = a\,(2n - 1) + b\,(n-1) + a\,n\,(n+1)/2 - a$

  Il s'en suit que $T(n) = \theta(n^2)$

On montre par ailleurs que le temps d'exécution reste en O(n.log(n)) quand le découpage opéré par la procédure partitionner s'effectue en gardant un rapport constant entre les tailles des deux partitions. On montre également que si les découpages performants et non performants alternent à chaque niveau de profondeur, le temps d'exécution du Quick Sort

reste toujours en O(n.log(n)) mais avec des constantes cachées sous la notation O légèrement supérieures.

Ainsi donc, le Quick Sort a une complexité en pire cas en $\theta(n^2)$ – cas où t[0..n-1] est initialement ordonné – mais une complexité ordinaire en O(n.log(n)). Les facteurs constants cachés dans la notation O restant faibles, le Quick Sort est souvent le meilleur algorithme de tri en pratique … si toutefois le risque du comportement en pire cas n'est pas rédhibitoire.

5.1.3.3 En pratique …

La différence d'efficacité entre $\theta(n^2)$ et O(n.log(n)) mérite des efforts pour réduire les risques de comportements en $\theta(n^2)$. Tout repose sur un choix pertinent de l'élément pivot par la procédure partitionner. L'idéal est bien sûr une valeur de pivot égale à la médiane de t[i..j], mais la détermination systématique de cette valeur est trop coûteuse car en O(j-i+1).

La procédure partitionner préalablement décrite choisit systématiquement comme valeur pivot l'élément t[i] initial. On pourrait choisir dans t[i..j] une autre valeur pivot, qu'il suffirait alors d'échanger avec t[i] avant d'appliquer la même procédure partitionner.

Afin de réduire le risque consécutif à des tableaux / sous-tableaux déjà ordonnés voire essayer de tendre vers un pivot médian, on peut envisager plusieurs approches, comme par exemple :

- choix stochastique : le pivot est une valeur choisie aléatoirement dans t[i..j]
- choix médian : le pivot est la valeur médiane d'un ensemble d'éléments de t[i..j], par exemple la valeur médiane de t[i], t[(i+j)/2] et t[j]
- choix stochastique médian : le pivot est la valeur médiane de 2p+1 éléments choisis aléatoirement dans t[i..j], où p est une constante entière

En outre, quand |j-i| devient petit (typiquement inférieur à 15, par exemple), des algorithmes de tri naïfs en $\theta(n^2)$ peuvent s'avérer plus efficaces car leurs complexités réelles sont plus petites. Le tri par insertion est généralement choisi car il est linéaire si le tableau est déjà ordonné.

Ainsi, un Quick Sort réel pourrait s'écrire :

```
// « Quick Sort amélioré » – Tri de t[i..j].
// Appel principal pour trier t[0..n-1] : QuickSort (t, 0, n-1).
Procédure QuickSort (t, i, j)
   si i<j alors
       si j-i<seuil alors
            TriParInsertion(t[i..j]) ;
       sinon
            i0 ← indice d'une valeur pivot dans t[i..j] ;
            échanger t[i] et t[i0] ;  // t[i] est l'élément pivot
            Partitionner(t, i, j, k) ;// t[i..k-1]≤t[k]<t[k+1..j]
            QuickSort (t, i, k-1) ;   // t[i..k-1]↑≤t[k]<t[k+1..j]
            QuickSort (t, k+1, j) ;   // t[i..k-1]↑≤t[k]<t[k+1..j]↑
       finSi ;
   finSi ;
   // t[i..j]↑
finProcédure
```

5.1.4 Tri par fusion (Merge Sort)

L'algorithme de tri par éclatement et fusion (split and merge) est du type « diviser pour régner ». Il ne trie pas sur place mais est stable et est particulièrement bien approprié au tri de listes.

Le principe de l'algorithme est : éclater l'ensemble d'éléments à trier en deux sous-ensembles de taille moitié, trier récursivement chacun de ces sous-ensembles, puis fusionner ces deux sous-ensembles triés en un ensemble trié global.

```
// « Tri par fusion » d'une liste chaînée d'éléments.
Procédure TriFusion (Liste L)
      Si | L | ≥ 2 alors
            Eclater(L, Li, Lp) ;    // éclate L en Li et Lp.
            // Li = liste des éléments de L de rang impair
            // Lp = liste des éléments de L de rang pair
            TriFusion(Li) ;
            // Li↑
            TriFusion(Lp) ;
            // Lp↑
            Fusion(Li, Lp, L) ;     // fusionne Li et Lp en L
            // L↑
      finSi ;
      // L↑
finProcédure
```

Soit n = |L|. L'éclatement de la liste L se réalise en $\theta(n)$. De même pour la fusion de Li et Lp. Le temps de calcul T(n) de cet algorithme est donc défini par la relation de récurrence :

$T(0) = T(1) = \theta(1)$

$T(n>1) = 2\ T(n/2) + \theta(n)$.

Le tri par fusion a donc une complexité en temps en $\theta(n.\log(n))$.

5.1.5 Tri par tas (Heap Sort)

Définitions :

- Dans un arbre, on dit qu'un nœud est **dominant** ssi sa clé est supérieure ou égale à celle de chacun de ses fils (une feuille étant un nœud trivialement dominant).
- Un **tas** est un arbre binaire presque parfait dont chaque nœud est dominant.

Propriétés :

- la clé maximale d'un tas est à la racine
- tout nœud d'un tas est lui-même racine d'un tas
- en tant qu'arbre binaire presque parfait, un tas de n nœuds est efficacement implémenté dans un tableau t[0..n-1] tel que :

  t[0] = nœud racine

  $\forall i \in [1..n-1]$ si t[i] = nœud i
  alors $2i+1<n \Rightarrow$ t[2i+1] = fils gauche de i
  $2i+2<n \Rightarrow$ t[2i+2] = fils droit de i

Nota : les éléments t[0..⌊n/2⌋-1] sont des nœuds internes ; les éléments t[⌊n/2⌋..n-1] sont des feuilles.

L'algorithme de tri par tas consiste à mettre le tableau t à trier sous la forme de tas puis à trier ce tas.

```
// « Tri par tas » d'un tableau t[0..n-1], n>0.
Procédure TriParTas (t[0..n-1])
      ConstruireTas(t[0..n-1]) ;
      // t[0..n-1] est un tas
      TrierTas(t[0..n-1]) ;
      // t[0..n-1]↑
finProcédure
```

5.1.5.1 Trier un tas

L'algorithme de tri d'un tas vérifie l'énoncé :

(t[0..n-1] est un tas ∧ n>0) { TrierTas(t[0..n-1]) } (t[0..n-1]↑)

```
// Tri d'un tas t[0..n-1], n>0.
Procédure TrierTas (t[0..n-1])
      // t[0..n-1] est un tas
      i ← n ;
      // I(i) = ( t[0..i-1] est un tas ∧ t[0..i-1]≤t[i..n-1]↑ )
      tant que i > 1 faire
            i ← i - 1 ;
            // I(i+1)
            permuter t[0] et t[i] ;
            // t[0..i-1]≤ t[i..n-1]↑ ∧ tout nœud autre que 0 de
            // l'arbre binaire presque parfait t[0..i-1] est
            // dominant.
            RétablirTas(t[0..i-1] ;
            // I(i)
      finTantQue ;
      // t[0..n-1]↑
finProcédure
```

5.1.5.2 Rétablir un tas

Soit à définir une procédure RétablirTas vérifiant l'énoncé :

(0≤j<k ∧ tout noeud autre que j du sous-arbre de racine j de l'abpp t[0..k-1] est dominant)

{ RétablirTas(t[j..k-1]) }

(j est racine d'un tas au sein de l'abpp t[0..k-1])

où « abpp » signifie « arbre binaire presque parfait ».

```
// Rétablir la propriété « t[j..k-1] est un tas », 0≤j<k, sachant que
// tout noeud autre que j du sous-arbre de racine j de l'arbre
// binaire presque parfait t[0..k-1] est dominant
Procédure RétablirTas (t[j..k-1])
      j' ← j ;
      // I(j') = tout noeud autre que j' du sous-arbre de racine j de
      // de l'arbre binaire presque parfait t[0..k-1] es dominant.
      tant que j' non dominant au sein de l'abpp t[O..k-1] faire
            jstar ← indice du fils de j' au sein de l'abpp t[0..k-1],
                    de clé maximale ;
            permuter t[jstar] et t[j'] ;
            // I(jstar)
            j' ← jstar ;
            // I(j')
      finTantQue ;
      // tout noeud du sous-arbre de racine j de l'abpp t[0..k-1] est
      // dominant,
      // donc j est racine d'un tas au sein de l'abpp t[0..k-1]
finProcédure
```

5.1.5.3 Construire un tas

La mise sous forme de tas d'un tableau t[0..n-1] vérifie l'énoncé :

$$(t[0..n-1]=[e_0, \ldots, e_{n-1}] \wedge n>0)$$

$$\{ \text{ConstruireTas}(t[0..n-1] \}$$

$$(t[0..n-1]=\prod(e_0, \ldots, e_{n-1}) \wedge t[0..n-1] \text{ est un tas})$$

```
// Mettre un tableau t[0..n-1], n>0, sous forme d'un tas.
Procédure ConstruireTas (t[0..n-1])
      i ← ⌊n/2⌋ ;
      // I(i) = tout nœud de [i..n-1] est dominant au sein de
      // l'arbre binaire presque parfait t[0..n-1].
      // I(i) est vrai à ce stade car toutes les feuilles sont
      // trivialement dominantes.
      tant i ≠ 0 faire
            i ← i-1 ;
            // I(i+1)
            RétablirTas(t[i..n-1]) ;
            // I(i)
      finTantQue ;
      // t[0..n-1] est un tas
finProcédure
```

Remarque : la procédure RétablirTas ne peut pas être appliquée de la racine vers les feuilles car sa pré-condition n'a aucune raison d'être vraie partout !

5.1.5.4 Complexité algorithmique

a) Complexité de la procédure RétablirTas(t[j..k-1]).

Au pire, la valeur initiale de t[j] chemine jusqu'à une feuille de l'arbre. La longueur parcourue est en $O(\lfloor \log_2(k-j) \rfloor)$. Cette longueur est le nombre d'itérations du corps de boucle, lui-même en temps unitaire constant. La procédure rétablirTas a donc une complexité en O(log(k-j)).

b) Complexité de la procédure ConstruireTas(t[0..n-1]).

Chaque appel de RétablirTas est en O(log(n)), et il existe θ(n) appels de ce type. Le temps d'exécution est donc en O(n.log(n)). Mais ce majorant, quoique correct, n'est pas asympotiquement très serré. On peut trouver un majorant plus fin en observant que le temps d'exécution de RétablirTas sur un noeud dépend de la hauteur h de ce noeud dans l'arbre et que la hauteur de la plupart des noeuds est faible. On s'appuie également sur les propriétés d'un tas de n éléments :

- la hauteur du tas est $\lfloor \log_2(n) \rfloor$
- le nombre de noeuds à la hauteur h est au plus de $\lceil n / 2^{h+1} \rceil$

Le temps requis par RétablirTas appelé sur un noeud de hauteur h étant O(h), le coût total de la procédure ConstruireTas est :

$$\sum_{h=0}^{hauteurTas} (\lceil n / 2^{h+1} \rceil \; O(h))$$

$$= O(n . \sum_{h=0}^{hauteurTas} (\lceil h / 2^{h} \rceil))$$

Sachant que lorsque $| x | < 1$:

$$\sum_{k=0}^{+\infty} (x^k) = x / (1-x)$$

On en déduit, en dérivant et multipliant par x, que :

$$\sum_{k=0}^{+\infty} (k \, x^k) = x / (1-x)^2$$

et donc en prenant x = 1/2 :

$$\sum_{k=0}^{+\infty} (k / 2^k) = 2$$

Le temps d'exécution de ConstruireTas peut donc être borné par :

$$O(n . \sum_{h=0}^{hauteurTas} (h / 2^h))$$

$$= O(n . \sum_{h=0}^{+\infty} (h / 2^h))$$

$$= O (n)$$

La procédure ConstruireTas a donc une complexité en O(n).

c) Complexité de la procédure TrierTas(t[0..n-1]).

La boucle de l'algorithme TrierTas exécute n-1 itérations. Chaque appel à RétablirTas(t[0..i-1]) est en $O(\log_2(i))$. Le coût total de la procédure TrierTas est donc en :

$$\theta(n) + \sum_{i=1}^{n-1} O(\log_2(i))$$

$$= \theta(n) + O(\log_2(\prod_{i=1}^{n-1} i))$$

$$= \theta(n) + O(\log_2((n-1)!))$$

Comme $n! = \sqrt{2\pi n}\,(\frac{n}{e})^n\,(1 + o(\frac{1}{n}))$ *[formule de Stirling[1]]*

il s'en suit que la procédure TrierTas est de complexité O(n.log(n)).

d) Complexité de la procédure TriParTas(t[0..n-1]).

Le tri par tas de t[0..n-1] a donc une complexité en O(n.log(n)).

5.1.5.5 En pratique …

Le tri par tas est un très bon algorithme mais le Quick Sort, bien implémenté, est généralement plus rapide en pratique. Le tri par tas, bien que plus lent en moyenne que le Quick Sort, présente toutefois deux avantages par rapport à ce dernier :

- son plus mauvais comportement est en O(n.log(n)), contre $\theta(n^2)$ pour le Quick Sort ;
- il requiert un espace mémoire additionnel constant (en O(1)), contre un espace mémoire additionnel en O(log(n)) pour la meilleure variante du Quick Sort.

Par ailleurs, la structure de tas est utile en elle-même. Les files de priorités en sont les principales applications, car les deux opérations de base : insérer et extraireMax s'exécutent en O(log(n)). Exemple d'application : la planification des tâches dans un ordinateur à ressources partagées.

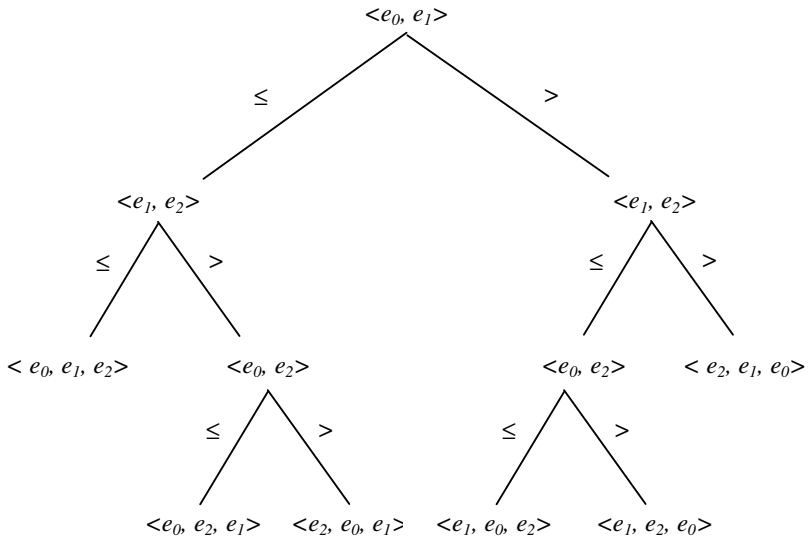
5.1.6 Optimalité des tris comparatifs

Un tri est une permutation croissante $\prod(<e_0, e_1, \ldots, e_{n-1}>)$. Exemple :

$\prod : <1, 7, 0, 8, 9\ 4> \mapsto <0, 1, 4, 7, 8, 9> \uparrow$

Il existe n! permutations différentes de la séquence $<e_0, e_1, \ldots, e_{n-1}>$. Le nombre de permutations élémentaires $e_i \leftrightarrow e_j$ nécessaires pour trier $<e_0, e_1, \ldots, e_{n-1}>$ donne une mesure du temps de calcul et donc de la complexité.

Exemple d'arbre de décision pour le tri de $<e_0, e_1, e_2>$:



[1] James STIRLING [1692-1770] est un mathématicien écossais. L'équivalent asymptotique de n!, pour lequel il est très connu, a été préalablement approché par le mathématicien français Moivre (Abraham de MOIVRE, 1667-1754, qui émigra en Angleterre en 1685 à la révocation de l'Edit de Nantes) sous la forme : $n! \approx C\ n^{n+1/2}\ e^{-n}$. C'est Stirling qui a démontré que $C = (2\pi)^{1/2}$.

La hauteur de l'arbre de décision donne une mesure de la complexité de l'algorithme de tri. Quelle est la hauteur h de l'arbre de décision pour le tri d'un ensemble de n éléments ?

Le nombre de feuilles de l'arbre de décision est au plus égal aux n! permutations. Or un arbre binaire quelconque de hauteur h comporte au plus 2^h feuilles. D'où :

$n! \leq 2^h$

Donc $h \geq \log_2(n\ !)$

Comme $n! = \sqrt{2\pi n}\,(\frac{n}{e})^n\,(1 + o(\frac{1}{n}))$ *[formule de Stirling]*

il s'en suit $n! \geq (\frac{n}{e})^n$

et donc $h \geq n.\log_2(n)$ en négligeant le terme $n.\log_2(e)$

c'est-à-dire $h = \Omega(n.\log_2(n))$.

Donc, tout algorithme de tri par comparaisons d'un ensemble de n éléments exige au moins $\Omega(n.\log(n))$ comparaisons. Et par conséquent, tout algorithme de tri comparatif en O(n.log(n)) est asymptotiquement optimal, c'est-à-dire qu'aucun autre algorithme de tri par comparaisons ne lui sera meilleur de plus d'un facteur constant.

5.2 Tris en temps linéaire

Les algorithmes de tri présentés dans les sections précédentes ne font aucune hypothèse sur l'entrée et reposent uniquement sur des comparaisons entre les éléments. Ce sont des tris par comparaisons. Or il a été montré (voir section précédente) que tout algorithme de tri comparatif en O(n.log(n)) – le Quick Sort par exemple – est asymptotiquement optimal. Comment donc envisager des algorithmes de tri qui s'exécutent en temps linéaire ? De tels algorithmes font des hypothèses sur l'entrée et font appel à des opérations autres que des comparaisons.

5.2.1 Tri par dénombrement

Soit t[0..n-1] tel que $\forall i\ 0 \leq t[i] < m$.

Soit à écrire un programme P répondant à l'énoncé :

$(n>0 \ \wedge \ \forall i \in [0..n-1]\ 0 \leq t[i] < m)\ \{ P \}\ (t'[0..n-1] = \prod(t[0..n-1]) \ \wedge \ t'[0..n-1]\uparrow)$

Si, par un programme P', on est capable de produire un tableau d[0..m-1] tel que :

$d[i] = \#(t[0..n-1] \leq i)$

c'est-à-dire d[i] = nombre d'éléments de t[0..n-1] de valeur inférieure ou égale à i

alors le programme P s'écrira :

```
P' ;
Pour i variant de n-1 à 0 faire d[t[i]] ← d[t[i]] – 1 ;  t'[d[t[i]]] ← t[i] ;  finPour ;
// t'[0..n-1]=∏(t[0..n-1]) ∧ t'[0..n-1]↑
```

Nota. La boucle sur i va de n-1 à 0 afin d'assurer la stabilité du tri.

Reste à écrire le programme P', qui répond à l'énoncé :

$(n>0 \ \wedge \ \forall i \in [0..n-1]\ 0 \leq t[i] < m)\ \{ P1 \}\ (\forall i \in [0..m-1]\ \ d[i] = \#(t[0..n-1] \leq i))$

```
// « Tri par dénombrement » d'un tableau t[0..n-1] à valeurs dans
// [0..m-1].
// Le tableau résultant trié est t'[0..n-1].

// ∀i∈[0..n-1]  0≤t[i]<m
Pour i variant de 0 à m-1 faire d[i] ← 0 ;  finpour ;
// ∀i∈[0..m-1]  d[i] = 0
Pour i variant de 0 à n-1 faire d[t[i]] ← d[t[i]] + 1 ;  finpour ;
// ∀i∈[0..m-1]  d[i] = #(t[0..n-1]=i)
Pour i variant de 1 à m-1 faire d[i] ← d[i] + d[i-1] ;  finpour ;
// ∀i∈[0..m-1]  d[i] = #(t[0..n-1]≤i)
Pour i variant de n-1 à 0 faire
      d[t[i]] ← d[t[i]] - 1 ;
      t'[d[t[i]]] ← t[i] ;
finPour ;
// t'[0..n-1]↑
```

Le tri par dénombrement est en $\theta(m+n)$ en temps et en $\theta(m+n)$ en mémoire auxiliaire.

6 ALGORITHMES DE SÉLECTION

Définition. Un élément d'un ensemble de n éléments est de rang i, i∈[1, n], ssi c'est le $i^{ème}$ plus petit élément de cet ensemble.

Exemples : l'élément de rang 1 est l'élément minimum et celui de rang n est l'élément maximum ; l'élément de rang $\lfloor (n+1)/2 \rfloor$ est l'élément médian inférieur (ou simplement élément médian si n est impair) et celui de rang $\lceil (n+1)/2 \rceil$ est l'élément médian supérieur.

Dans ce chapitre, on se pose le problème de déterminer quel est l'élément de rang i, i∈[1, n], d'un tableau de n éléments distincts, n>0.

6.1 Sélection des extrema

Soit à écrire un programme P répondant à l'énoncé :

(n>0) { P } (xmin = min(t[0..n-1]) ∧ xmax = max(t[0..n-1]))

La solution intuitive qui consiste à comparer chaque élément de t avec chacun des candidats xmin et xmax courants, conduit au total à 2(n-1) comparaisons en pire cas. Mais, bien que ce nombre soit asymptotiquement optimal, il est possible de faire mieux. On peut en effet ne réaliser que $3\lfloor n/2 \rfloor$ comparaisons en traitant les éléments deux à deux et en commençant à les comparer entre eux avant de les comparer à l'un ou l'autre des candidats xmin ou xmax.

```
// Sélection concomitante du minimum et du maximum de t[0..n-1], n>0.
Si n impair alors   xmin ← t[0] ; xmax ← t[0] ; i ← 1 ;
sinon si t[0]<t[1]
            alors   xmin ← t[0] ; xmax ← t[1] ; i ← 2 ;
            sinon   xmin ← t[1] ; xmax ← t[0] ; i ← 2 ;
      finSi ;
finSi ;
// I(i, xmin, xmax) = ( xmin=min(t[0..i-1])  ∧  xmax=max(t[0..i-1]) ∧
//                      n-i pair )
Tant que i < n faire
      // I(i, xmin, xmax) ∧ i≤n-2
      si t[i] < t[i+1] alors
            si t[i]   < xmin alors xmin ← t[i] ;   finsi ;
            si t[i+1] > xmax alors xmax ← t[i+1] ; finsi ;
      sinon
            si t[i+1] < xmin alors xmin ← t[i+1] ; finsi ;
            si t[i]   > xmax alors xmax ← t[i] ;   finsi ;
      finsi ;
      // I(i+2, xmin, xmax) ∧ i≤n-2
      i ← i+2 ;
      // I(i, xmin, xmax) ∧ i≤n
finTantQue ;
// xmin=min(t[0..n-1])  ∧  xmax=max(t[0..n-1])
```

6.2 Sélection de l'élément de rang i

Soit à écrire un programme P répondant à l'énoncé :

(n>0 ∧ i∈[1, n]) { P } (x = élément de rang i de t[0..n-1])

Pour solutionner ce problème, on peut évidemment trier t[0..n-1] par valeurs croissantes avec un algorithme de tri efficace en O(n.log(n)) : l'élément cherché sera alors t[i-1]. Mais il s'avère qu'il n'est pas nécessaire de trier complètement le tableau t pour pouvoir résoudre le problème, ce qui conduira à une complexité algorithmique moindre.

6.2.1 Sélection en temps moyen linéaire

L'algorithme, de type « diviser pour régner », s'inspire de l'algorithme du Quick Sort : il partitionne le tableau t autour d'un élément pivot puis s'applique récursivement sur la partie de la partition qui contient l'élément cherché.

```
// Sélection de l'élément de rang i de t[p..r], p≤r, i∈[1, r-p+1],
// en temps moyen linéaire

Fonction Selection (t, p, r, i) : Element
   si p = r  alors retourner t[p] ; finSi ;
   // sinon
   p0 ← indice aléatoire à valeur dans [p..r] ;
   échanger t[p] et t[p0] ;     // t[p] est l'élément pivot
   Partitionner(t, p, r, q) ;   // t[p..q-1]≤t[q]<t[q+1..r]
   k ← q - p + 1 ;              // k = nb d'éléments de t[p..q]
   si i = k alors            retourner t[q] ;
   sinon si i < k alors      retourner Selection(t, p, q-1, i) ;
         sinon               retourner Selection(t, q+1, r, i-k) ;
finFonction
```

Quelle est la complexité de cet algorithme en fonction du nombre n d'éléments de t[p..q] ? Comme pour le Quick Sort, le cas le plus défavorable est en $\theta(n^2)$, mais, en pratique, le choix stochastique du pivot limite les risques d'un tel cas.

Quelle est la complexité de cet algorithme en moyenne ? Notons T(n) le temps de calcul de l'algorithme. T(n) est une variable aléatoire dont nous allons majorer l'espérance E[T(n)].

Notons X_k la variable aléatoire indicatrice définie par :

$X_k = 1$ si le sous-tableau t[p..q] a exactement k éléments
$X_k = 0$ sinon

L'espérance $E[X_k]$ est l'espérance que le pivot soit le $k^{ième}$ plus petit élément de t[p..r]. Tous les éléments ayant une chance équiprobable d'être choisis comme pivot, cette espérance est donc de 1/n.

En supposant T(n) monotone croissante, on peut borner le temps de l'appel récursif par le temps de l'appel récursif du plus grand des deux sous-tableaux de la partition.

Ainsi donc :

$$T(n) \le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

Donc :

$$E[T(n)] \leq E[\sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))]$$

L'espérance étant linéaire et les variables aléatoires X_k et T(n) indépendantes, il s'en suit :

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n)$$

Les termes $T(\lceil n/2 \rceil)$ à T(n-1) apparaissant deux fois dans la sommation, et, si n est impair, $T(\lfloor n/2 \rfloor)$ apparaissant une fois, on obtient :

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n)$$

Prouvons par récurrence mathématique que E[T(n)] = O(n), c'est-à-dire qu'il existe une constante c telle que, pour toute valeur de n supérieure ou égale à une certaine constante n_0, $E[T(n)] \leq cn$.

Supposons cette propriété vraie $\forall k < n$. Alors, dans le cas général :

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an$$

c et a étant des constantes. On en déduit facilement que :

$$E[T(n)] \leq (3cn)/4 + c/2 - (2c)/n + an$$
$$\leq (3cn)/4 + c/2 + an$$
$$= cn - (cn/4 - c/2 - an)$$

Cette dernière expression est au plus égale à cn si $(cn/4 - c/2 - an) \geq 0$, c'est-à-dire dès que $n \geq 2c / (c - 4a)$ si on choisit c tel que $c - 4a > 0$. Ainsi, par exemple, si on choisit c=8a, alors $n_0 = 4$.

Pour terminer la preuve par récurrence, il reste à prouver que la propriété T(n) = O(n) reste valable pour les conditions aux limites, c'est-à-dire quand n < 4. Dans ces cas, T(n) = O(1), et donc, compte tenu que $n \geq 1$ par hypothèse, T(n) = O(n). Il suffit en effet alors de choisir $c \geq \max(8a, \max_{i \leq k \leq 4} (T(k)/k))$.

Donc E[T(n)] = O(n).

L'algorithme considéré a une complexité en temps en O(n) en moyenne. Les constantes cachées étant faibles, le cas quadratique étant peu probable, c'est un bon algorithme en pratique.

6.2.2 Sélection en temps linéaire en pire cas

L'algorithme précédent est efficace en pratique, mais en moyenne seulement. Cette section présente un algorithme linéaire en temps en pire cas, mais dont les constantes cachées sont beaucoup plus grandes et en font, finalement, un algorithme d'intérêt pratique moindre.

L'approche algorithmique reste la même que précédemment, mais un très gros effort est mis sur le choix du pivot.

```
// Sélection de l'élément de rang i de t[p..r], p≤r, i∈[1, r-p+1],
// en temps linéaire en pire cas
Fonction Selection (t, p, r, i) : Element
   si p = r  alors retourner t[p] ; finSi ;
   // sinon
   a) Diviser les n éléments de t[p..r] en ⌊n/5⌋ paquets de taille 5,
   plus éventuellement un dernier paquet constitué des n mod 5
   éléments restants ;
   b) Déterminer l'élément médian de chaque paquet de la façon
   suivante : trier par insertion le paquet puis sélectionner le
   3ème élément ;
   c) Déterminer le médian M des ⌈n/5⌉ médians déterminés à l'étape
   précédente, par un appel récursif de la fonction Selection
   d) En notant p0 l'indice de M,
   échanger t[p] et t[p0] ;     // t[p] est l'élément pivot
   Partitionner(t, p, r, q) ;   // t[p..q-1]≤t[q]<t[q+1..r]
   k ← q - p + 1 ;              // k = nb d'éléments de t[p..q]
   si i = k alors             retourner t[q] ;
   sinon si i < k alors       retourner Selection(t, p, q-1, i) ;
         sinon                retourner Selection(t, q+1, r, i-k) ;
finFonction
```

Quelle est la complexité de cet algorithme en fonction du nombre n d'éléments de t[p..r] ?

Soit n' le premier multiple de 5 impair supérieur ou égal à n. On a : $n \le n' = 5(2m+1) \le n+9$. Rajoutons n' – n éléments de valeur $+\infty$ à l'ensemble des valeurs, dorénavant de taille n'. Nous avons donc maintenant 2m+1 paquets de 5 éléments. La liste ordonnée des médianes de ces paquets est :

$$M_1 < \ldots < M_m < M < M_{m+2} < \ldots < M_{2m+1}$$

l'inégalité stricte résultant de l'hypothèse initiale que tous les éléments de t sont distincts.

Quel que soit le paquet $j \in [1, m]$, il y a au moins 3 éléments de ce paquet qui sont inférieurs à M : ceux qui sont inférieurs ou égaux à M_j. Dans le paquet m+1, il y a 2 exactement éléments inférieurs à M. Donc au total, il y a dans t[p..r] au moins 3m+2 éléments inférieurs à M. De la même façon, il y a au moins 3m+2 éléments supérieurs à M.

Le coût T(n) de l'algorithme est donc :

$$T(n) = T(n'/5) + T(n' - (3m+2)) + \theta(n)$$

où $T(n'/5)$ est le temps de calcul du médian des médians

$T(n' - (3m+2))$ est le temps de l'appel récursif

$\theta(n)$ est le temps de calcul de la médiane de chaque paquet (en $\theta(n)$ car en $\theta(1)$ sur chacun des $\lceil n/5 \rceil$ paquets) et de la partition de t autour du pivot M.

Compte tenu que :

- $n' = 5(2m+1) \le n+9$
- $n' - (3m+2) \le n' - \frac{3}{2}\left(\frac{n'}{5} - 1\right) - 2 \le \frac{7}{10}(n+9) - \frac{1}{2} < \frac{7}{10}n + 6$

et en supposant T(n) monotone croissante, on peut borner supérieurement T(n) :

$T(n) \leq T((n+9)/5) + T(\frac{7}{10} n + 6) + \theta(n)$

Prouvons par récurrence mathématique que $T(n) = O(n)$, c'est-à-dire qu'il existe une constante c telle que, pour toute valeur de n supérieure ou égale à une certaine constante n_0, $T(n) \leq cn$.

Supposons cette propriété vraie $\forall k < n$. Alors :

$$T(n) \leq c(n+9)/5 + c(\tfrac{7}{10} n + 6) + an$$
$$= \tfrac{9}{10} cn + \tfrac{39}{5} c + an$$
$$= cn - (\tfrac{1}{10} cn - \tfrac{39}{5} c - an)$$

Cette expression est au plus égale à cn ssi $\frac{1}{10} cn - \frac{39}{5} c - an \geq 0$, c'est-à-dire dès que $n \geq 78c / (c - 10 a)$ si on choisit c tel que $c - 10a > 0$. Ainsi, par exemple, si on choisit c=20a, alors $n_0 = 156$.

Pour terminer la preuve par récurrence, il reste à prouver que la propriété $T(n) = O(n)$ reste valable pour les conditions aux limites, c'est-à-dire quand $n < 156$. Dans ces cas, $T(n) = O(1)$, et donc, compte tenu que $n \geq 1$ par hypothèse, $T(n) = O(n)$. Il suffit en effet alors de choisir $c \geq \max(20a, \max_{i \leq k \leq 156} (T(k))/k))$.

Donc l'algorithme considéré est en O(n) dans le cas le plus défavorable.

Remarques :

- Si on découpe t en paquets de 3 au lieu de 5, alors :

  $n \leq n' = 3(2m+1) \leq n+5$

  $T(n) = T(n'/3) + T(n' - (2m+1)) + \theta(n)$

  donc $T(n) = T(n'/3) + T(2n'/3) + \theta(n)$

  ce qui conduit à un temps en O(n log(n)) !!

- Si on découpe t en paquets de 7 ou plus, l'algorithme demeure en O(n).

7 PROGRAMMATION DYNAMIQUE

La méthode « **diviser pour régner** » s'applique quand il est possible de diviser le problème en sous-problèmes indépendants et d'en combiner les résultats de façon raisonnablement efficace. Par contre, elle ne conduit pas à des algorithmes efficaces quand les sous-problèmes ne sont pas indépendants et ont eux-mêmes des sous-problèmes communs.

La **programmation dynamique** – inventée par Bellman[1] – est une méthode de conception à considérer quand :

- une solution optimale du problème est composée de solutions optimales de sous-problèmes de même nature (principe d'optimalité de Bellman)
- les sous-problèmes se recouvrent

La programmation dynamique consiste à éliminer les recalculs dans un programme récursif en mémorisant les résultats produits pour les valeurs d'arguments déjà calculés, tout au moins pour les plus petites valeurs de ces arguments. Dans certains cas, la solution récursive peut même être abandonnée au profit d'une solution itérative.

La programmation dynamique classique, dite **bottom-up** (ascendante), consiste à modifier l'algorithme pour construire systématiquement un tableau de valeurs en travaillant par taille de problème croissante. La programmation dynamique **top-down** (descendante) consiste à mémoriser les résultats intermédiaires au fil des appels récursifs (on parle alors de fonction à mémoire).

Exemple avec la suite de Fibonacci définie par la récurrence :

$$fibo(0) = 0$$
$$fibo(1) = 1$$
$$fibo(n>1) = fibo(n-1) + fibo(n-2)$$

- *Calcul par application directe de la méthode « diviser pour régner » :*

```
Fonction fibo(Naturel n) : Naturel
      si n<2 alors retourner n ;
      retourner fibo(n-1) + fibo(n-2) ;
finFonction
```

  Cet algorithme est très inefficace car les multiples appels récursifs conduisent à calculer les mêmes fibo(i) de multiples fois. Cet algorithme est exponentiel.

- *Calcul par programmation dynamique bottom-up*

```
Fonction fibo(Naturel n) : Naturel
      Soit t[0..n] un tableau
```

[1] Richard E. BELLMAN (1920-1984) est un mathématicien américain. Il est surtout célèbre comme inventeur de la programmation dynamique. Pour la petite histoire, Bellman a choisi le terme « programmation dynamique » dans un souci de communication. Son supérieur ne voulant ni du mot « recherche » ni du mot « mathématique », il lui a semblé que les termes « programmation » et « dynamique » seraient plus « vendeurs ». A noter qu'à l'époque, « programmation » avait davantage le sens de « planification » ou « ordonnancement » que celui qu'il a communément en informatique de nos jours.

```
        t[0] ← 0 ;
        t[1] ← 1 ;
        pour i variant de 2 à n faire
              t[i] ← t[i-1] + t[i-2] ;
        retourner t[n];
finFonction
```

Cet algorithme est en $\theta(n)$ en temps et en $\theta(n)$ en mémoire auxiliaire.

- *Calcul par programmation dynamique top-down*

```
// fonction à mémoire
Fonction fibo(Naturel n) : Naturel
        // f est une table statique mémorisant les fibo(i) déjà
        // calculés.
        si f[n] connu alors retourner f[n] ;
        si n<2 alors retourner n ;
        f[n] ← fibo(n-1) + fibo(n-2) ;
        retourner f[n] ;
finFonction
```

Cet algorithme est en $\theta(n)$ en temps si la détermination de f[n] est en $\theta(1)$, et en $\theta(n)$ en mémoire auxiliaire.

Cet exemple pourrait laisser penser que l'approche bottom-up est la panacée pour les problèmes relevant de la programmation dynamique. Il n'en est évidemment rien. De façon générale, la programmation dynamique bottom-up peut en effet calculer des valeurs inutiles, car rien ne garantit que les valeurs calculées serviront toutes ensuite. Par contre dans l'approche top-down, la fonction à mémoire, guidée par le besoin, ne calcule que les valeurs nécessaires. Mais les appels récursifs non terminaux sont consommateurs de temps et de mémoire …

7.1 Programmation dynamique appliquée aux problèmes d'optimisation

La programmation dynamique est une méthode tabulaire exacte de résolution de problèmes d'optimisation séquentielle dont la fonction objectif est monotone croissante et exprime qu'une solution optimale du problème est composée de sous-solutions optimales.

Résoudre un problème d'optimisation, c'est chercher une solution optimale par rapport à un certain coût (ou fonction objectif). Les algorithmes de programmation dynamique suivent typiquement le schéma générique le suivant :

1) Construire une table des coûts, qui, in fine, délivrera le coût d'une solution optimale. Le coût de la solution n'étant pas nécessairement la solution du problème, cette table est généralement enrichie lors de sa construction avec des informations complémentaires qui permettront ensuite de reconstruire la solution optimale cherchée. On appelle parfois cette table enrichie « table de remontée ».
2) Parcourir cette table en sens inverse de sa construction, du problème principal vers les sous-problèmes : la solution cherchée correspond au chemin parcouru.

De très nombreux problèmes relèvent de la programmation dynamique. *Exemples : triangulation optimale de polygones, recherche du plus long sous-mot commun à deux chaînes de caractères, multiplication d'une suite de matrices, problème du sac à dos, détermination du plus court chemin dans un graphe, …*

7.2 Exemple : distance d'édition

On se pose le problème de la mesure de la similarité entre deux chaînes de caractères. Le degré de similarité sera mesuré par une distance – appelée distance d'édition (ou distance de Levenshtein) – définie par le nombre minimal d'opérations d'édition nécessaires pour transformer une chaîne de caractères en l'autre.

Considérons deux chaînes de caractères x[0..m-1] et y[0..n-1], m>0, n>0. Le tableau suivant considère trois grands types de différences entre x et y et, pour chacun d'eux, propose une opération d'édition pour réduire cette différence.

Types de différences entre x et y	**Opérations d'édition sur x permettant de réduire la différence entre x et y**
Deux caractères de même position sont différents	*Remplacer* le caractère de x par celui de y
Un caractère de y manque dans x à la même position	*Insérer* ce caractère dans x à cette position
Un caractère de x manque dans y à la même position	*Supprimer* ce caractère de x à cette position

On notera que les opérations insérer et supprimer sont duales : insérer un caractère de y dans x à la même position est équivalent, du point de vue de la distance d'édition, à supprimer le caractère de y à cette position.

Exemple

Soit x = "ABCDE" et y = "ACHHE". La transformation de x en y peut se réaliser par la séquence d'opérations suivante (qui s'avère être de longueur minimale) : supprimer B, remplacer D par H, insérer H.

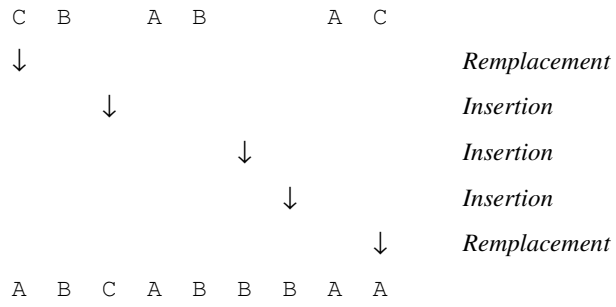
```
A  B  C  D     E
   ↓                 Suppression
         ↓           Remplacement
            ↓        Insertion
A     C  H  H  E
```

Exemple

Soit x = "CBABAC" et y = "ABCABBBAA". La transformation de x en y peut se réaliser, par exemple, par la séquence d'opérations suivante :

```
      C  B  A  B        A  C
↓                                Insertion
   ↓                             Insertion
         ↓                       Suppression
               ↓                 Insertion
                  ↓              Insertion
                     ↓           Remplacement
A  B  C     A  B  B  B  A  A
```

ou aussi par la séquence suivante (de longueur minimale) :



Le problème que l'on cherche à résoudre est de déterminer le plus petit nombre d'opérations élémentaires (remplacer / insérer / supprimer un caractère à un indice donné) pour transformer x en y.

Exemples

> *Il faut au moins 3 opérations d'édition pour transformer "CARIE" en "DURITE".*
>
> *Il faut au moins 4 opérations d'édition pour transformer "ALUMINIUM" en "ALBUMINE".*

Soit edit(x[0..m-1], y[0..n-1]) la distance d'édition entre x[0..m-1] et y[0..n-1]. On conviendra de la noter plus simplement et de façon équivalente : edit(m, n).

- Si x[m-1] = y[n-1], alors edit(m, n) = edit(m-1, n-1).
- Si x[m-1] ≠ y[n-1], alors trois possibilités d'opération virtuelle s'offrent à nous :
  - soit supprimer x[m-1] : auquel cas edit(m, n) = edit(m-1, n) + 1
  - soit insérer y[n-1] en x[m] : auquel cas edit(m, n) = edit(m, n-1) + 1
  - soit remplacer x[m-1] par y[n-1] : auquel cas edit(m, n) = edit(m-1, n-1) + 1

Mais il n'est pas possible par un simple test local de déterminer l'opération (ou une opération) qui conduira, in fine, à une solution optimale. La distance d'édition se définit donc par la relation de récurrence suivante :

edit(0, n) = n // car il suffit alors de réaliser n insertions

edit(m, 0) = m // car il suffit alors de réaliser m suppressions

edit(m, n) = min(edit(m-1, n)+1, edit(m, n-1)+1, edit(m-1, n-1)+ δ(x[m-1], y[n-1])

où $\delta(a, b) = 0$ si a = b, ou 1 sinon.

Ce problème, qui vérifie le principe de Bellman, relève clairement d'un problème de programmation dynamique. La solution itérative bottom-up met en oeuvre une table edit[0..m, 0..n] définie par :

edit[0, j] = j

edit[i, 0] = i

edit[i, j] = min(edit[i-1, j]+1, edit[i, j-1]+1, edit[i-1, j-1]+ δ(x[i-1], y[j-1])

```
// Calcul de la distance d'édition de x[0..m-1] et y[0..n-1].
Fonction distanceEdition(x[0..m-1], y[0..n-1]) : entier naturel
      soit edit[0..m, 0..n] une matrice d'entiers ;
      pour i variant de 0 à m faire edit[i, 0] ← i ; finPour ;
      pour j variant de 1 à n faire edit[0, j] ← j ; finPour ;
      pour i variant de 1 à m faire
            pour j variant de 1 à n faire
                  edit[i, j] ← min(edit[i-1, j] + 1,
                                   edit[i, j-1] + 1,
                                   edit[i-1, j-1] + δ(x[i-1],y[j-1]));
            finPour ;
      finPour ;
      retourner edit[m, n] ;
finFonction
```

Cet algorithme est en $\theta(m.n)$ en temps et en $\theta(m.n)$ en mémoire auxiliaire.

A noter que l'algorithme n'a pas vraiment besoin de mémoriser totalement la matrice edit pour calculer la distance d'édition : deux colonnes glissantes suffisent. La mémorisation complète de la matrice est toutefois nécessaire si l'on désire déterminer la séquence des opérations d'édition transformant x en y.

7.3 Exemple : multiplication d'une suite de matrices

On se pose le problème de calculer efficacement le produit de plusieurs matrices de dimensions variables. On suppose des matrices de nombres (entiers ou réels).

Soient n matrices M_0, M_1, …, M_{n-1}, chaque matrice M_i ayant m_i lignes et m_{i+1} colonnes. Soit à calculer la matrice produit : $M = M_0 M_1 \ldots M_{n-1}$

Le problème consiste à trouver l'ordre des produits à réaliser de façon à minimiser le nombre total d'opérations.

On rappelle que le calcul de $M_{i-1} M_i$ nécessite $O(m_{i-1} m_i m_{i+1})$ opérations[1].

Exemple

> *Considérons les matrices M_1, M_2, M_3, de tailles respectives 50×10, 10×100, 100×5.*
>
> *Le calcul $M_1 M_2$ nécessite 50000 multiplications ; le produit du résultat par M_3 demande 5000 multiplications. Soit un total de 55000.*
>
> *Le calcul $M_2 M_3$ nécessite 5000 multiplications ; le produit du résultat par M_1 demande 2500 multiplications. Soit un total de 5250.*
>
> *Il est donc bien plus efficace de calculer $(M_1(M_2 M_3))$ que $((M_1 M_2)M_3)$.*

[1] Quelle est la complexité du produit de deux matrices $n \times n$? Un algorithme naïf demande $\theta(n^3)$ opérations. Le meilleur algorithme général connu, celui COPPERSMITH-WINOGRAD, demande $O(n^{2.376})$ opérations, mais, compte tenu des grandes constantes cachées dans la notation O, n'est pas très utilisable. En pratique, pour des valeurs de n assez grandes (de l'ordre de quelques dizaines), le seul algorithme rapide véritablement utilisable est l'algorithme de STRASSEN, qui est en $\theta(n^{\log_2(7)})$. Quand les matrices sont peu denses voire creuses (beaucoup de zéros), il existe des algorithmes spécifiques dont la complexité est de l'ordre de $O(n^2)$.

Notons c(i, j) le nombre minimum de *multiplications* nécessaires pour calculer $M_i\ M_{i+1}\ldots M_j$:

- $c(i, i) = 0$ // cas 1 seule matrice : i=j
- Dans le cas i<j, il s'agit de choisir un $k \in [i, j-1]$ qui minime le nombre d'opérations du produit $(M_i\ M_{i+1}\ldots M_k)(M_{k+1}\ldots M_j)$. D'où :

  $c(i, j) = \min_k(c(i, k) + c(k+1, j) + m_i\ m_{k+1}\ m_{j+1})$ // cas général : i<j

Déterminer l'ordre optimal d'évaluation du produit $M_i\ M_{i+1}\ldots M_j$ revient donc, dans le cas général i<j, à :

- chercher le meilleur k qui minimise les $c(i,k) + c(k+1,j) + m_i\ m_{k+1}\ m_{j+1}$
- déterminer l'ordre optimal d'évaluation des sous-produits $(M_i\ M_{i+1}\ldots M_k)$ et $(M_{k+1}\ldots M_j)$.

Un algorithme récursif qui traduirait simplement la relation de récurrence précédente serait particulièrement inefficace. Le nombre total $n_{i,j}$ d'appels récursifs nécessaires suivrait en effet une récurrence à deux indices :

$$n_{i,j} = \sum_k\ n_{i,k} + n_{k+1,j}$$

et serait considérable.

La programmation dynamique évite cette explosion. La solution itérative bottom-up de ce problème met en oeuvre deux tables triangulaires :

- Une table c[0..n-1, 0..n-1] destinée à mémoriser les différents c(i, j) évalués, évitant ainsi de les recalculer plusieurs fois. La table c[0..n-1, 0..n-1] est définie par :

  $c[i, i] = 0$

  $c[i, j] = \min_k(c[i, k] + c[k+1, j] + m_i\ m_{k+1}\ m_{j+1})$ $\quad \forall\ 0 \le i < j \le n-1$

  In fine, c[0, n-1] sera le nombre minimum de multiplications nécessaire pour calculer $M_0\ M_1\ldots M_{n-1}$.

- Une table k[0..n-1, 0..n-1] destinée à mémoriser les k optimaux déterminés dans les calculs des c[i, j] :

  $k[i, j]$ = valeur de k qui minime $c[i, k] + c[k+1, j] + m_i\ m_{k+1}\ m_{j+1}$

  = valeur k d'évaluation optimale de $M_i\ M_{i+1}\ldots M_j$ par $(M_i\ M_{i+1}\ldots M_k)(M_{k+1}\ldots M_j)$.

La programmation itérative de cette récurrence nécessite d'être attentif à avoir bien calculé les c[i, k] et c[k+1, j] avant d'en avoir besoin ! Pour cela il faut les calculer dans l'ordre A, B, C, ... F indiqué sur le schéma suivant (exemple d'une matrice c[0..3, 0..3]) :

	0	1	2	3
0	0	A	C	F
1		0	B	E
2			0	D
3				0

L'algorithme, sa complexité, et un exemple sont donnés ci-après.

```
// Détermination et affichage de l'ordre optimal d'évaluation du
// produit de matrices M0 M1 … Mn-1.

// Paramètre : la matrice m[0..n] des tailles définie par :
//    ∀ i∈[0..n-1] m[i] = nombre de lignes de Mi
//    ∀ i∈[1..n]   m[i] = nombre de colonnes de Mi-1

Procédure ordreEval( Matrice m[0..n] ) ;
      soit c[0..n-1, 0..n-1] une matrice d'entiers naturels ;
      soit k[0..n-1, 0..n-1] une matrice d'entiers naturels ;

      c[0, 0] = 0 ;
      pour j variant de 1 à n-1 faire
            c[j, j] = 0 ;
            pour i variant de j-1 à 0 par pas de -1 faire
                  cmin ← +∞ ;
                  pour k variant de i à j-1 faire
                        c ← c[i,k] + c[k+1,j] + m[i]*m[k+1]*m[j] ;
                        si c < cmin alors
                              cmin ← c ;
                              kstar ← k ;
                        finSi ;
                  finPour ;
                  c[i, j] ← cmin ;
                  k[i, j] ← kstar ;
            finPour ;
      finPour ;
      afficherOrdreEval(k[0..n-1, 0..n-1], 0, n-1) ;
finFonction
```

```
// Affichage de l'ordre optimal d'évaluation du produit de
// matrices Mi Mi+1 … Mj

// Paramètre : la matrice k[0..n-1, 0..n-1] définie par :
//    k[i, j] = valeur k d'évaluation optimale de Mi Mi+1 … Mj
//    par (Mi Mi+1 … Mk)(Mk+1 … Mj)

Procédure afficherOrdreEval( Matrice k[0..n-1, 0..n-1],
                             entier i, entier j ) ;
      si i = j alors
            imprimer("Mi")
      sinon
            k ← k[i, j] ;
            imprimer "(" ;
            afficherOrdreEval(k[], i, k) ;
            afficherOrdreEval(k[], k+1, j) ;
            imprimer ")" ;
finProcédure
```

Le calcul de la matrice c[0..n-1, 0..n-1] est en $O(n^3)$ car il y a n(n+1)/2 coûts à calculer et, pour chaque coût, il faut minimiser une expression d'au plus n termes. Quant à l'algorithme récursif de reconstruction de l'ordre des évaluations, il est en $\theta(n)$. Au final, l'algorithme de détermination de l'ordre optimal d'évaluation d'un produit de matrices $M_0 M_1 \ldots M_{n-1}$ a donc une complexité en temps de $O(n^3)$.

Exemple :

Soit à calculer le produit des 6 matrices $M_0 M_1 \ldots M_5$, leurs tailles respectives étant définies par :

m[0..6] =

0	1	2	3	4	5	6
30	35	15	5	10	20	25

La matrice c[0..5, 0..5] sera :

	0	1	2	3	4	5
0	0	15750	7875	9375	11875	15125
1		0	2625	4375	7125	10500
2			0	750	2500	5375
3				0	1000	3500
4					0	5000
5						0

La matrice k[0..5, 0..5] sera :

	0	1	2	3	4	5
0	-	0	0	2	2	2
1		-	1	2	2	2
2			-	2	2	2
3				-	3	4
4					-	4
5						-

L'ordre optimal d'évaluation sera : $((M_0 (M_1 M_2))((M_3 M_4) M_5)$

8 ANNEXE A – Rappels mathématiques

8.1 Logique

Opérateurs logiques	
Et	$\wedge$
Ou	$\vee$
Non	$\neg$
Implication	$\Rightarrow$
Equivalence	$\Leftrightarrow$

L'usage répandu en mathématiques est de n'écrire que des propositions vraies. Ainsi quand on écrit $p \Rightarrow q$ on signifie $(p \Rightarrow q) =$ vrai.

8.1.1 Sémantique des connecteurs ET et OU

Les opérateurs ET et OU de la logique formelle n'ont pas nécessairement la même sémantique que ces mêmes connecteurs en langage naturel.

Le ET logique ($\wedge$) est commutatif, alors que le ET en langage naturel ne l'est pas forcément car il peut sous-tendre un lien de cause à effet.

Exemple : « Il prit peur et le tua » vs « Il le tua et prit peur »

Le OU logique ($\vee$) est inclusif, alors que le OU en langage naturel est généralement exclusif.

Exemple : « Parisien ou provincial », « Fromage ou dessert »

8.1.2 Implication logique – Equivalence logique

L'implication en langage naturel s'exprime par un énoncé du type « Si hypothèse alors conclusion ». Elle indique une relation de causalité entre hypothèse et conclusion.

L'implication logique est le connecteur binaire qui, à deux propositions p et q, associe la proposition $p \Rightarrow q$ (lire « p implique q ») qui n'est qu'une notation commode pour exprimer l'alternative $\neg p \vee q$ sans qu'il y ait de relation de cause à effet exigée entre p et q.

Table de vérité de $p \Rightarrow q$		
p	**q**	**$p \Rightarrow q$**
F	F	V
F	V	V
V	F	F
V	V	V

L'implication $p \Rightarrow q$ peut également se lire : « si p est vrai alors q est vrai ».

Dans la littérature, l'implication logique est aussi souvent notée : $p \supset q$

L'opérateur d'implication logique n'a pas une signification intuitive car il faut se convaincre que « faux implique tout » (faux $\Rightarrow$ *)[1], que « tout implique vrai » (* $\Rightarrow$ vrai), et que $p \Rightarrow q$ peut être vraie bien que p puisse ne pas être la cause de q.

Propriétés :

- $\{ \neg, \Rightarrow \}$ constitue un ensemble d'opérateurs complet, c'est-à-dire une base. Toute proposition logique peut donc être exprimée uniquement avec les opérateurs $\neg$ et $\Rightarrow$.
- $(p \Rightarrow q) = (\neg q \Rightarrow \neg p)$
- $(p \Rightarrow q) = (p \wedge q) \vee \neg p$

  Cette propriété est importante car, lorsqu'on souhaite traduire en logique formelle un énoncé, on se trouve souvent à devoir choisir entre les connecteurs $\Rightarrow$ et $\wedge$. Cette propriété met bien en évidence la différence sémantique entre les deux opérateurs.
- $\neg (p \Rightarrow q) = (p \wedge \neg q)$

Exemple

Considérons un tableau t[0..n-1] de n éléments, n>0. Pour un x donné, soit P(i) la propriété « tout élément de t d'indice strictement inférieur à i est différent de x ». Cet énoncé a sens pour tout i appartenant [1, n] car alors la notion d'« élément de t d'indice strictement inférieur à i » existe. Par contre, pour i=0, cette notion n'existe pas. Et pourtant P(0) a sens et est vraie car, dans le cas i=0, il est vrai que x n'est pas élément de l'ensemble des éléments de t d'indice strictement inférieur à i, puisque cet ensemble est vide.

Comment traduire formellement P(i) ?

Formulation 1 : $P_1(i) = (i \in [1, n] \wedge x \notin t[0..i-1])$

Cette formulation est incorrecte car $P_1(0)$ = faux.

Formulation 2 : $P_2(i) = (i \in [0, n] \wedge x \notin t[0..i-1])$

Cette formulation est acceptable sous hypothèse de donner sens à t[0..-1] en considérant que t[0..i-1] représente un ensemble vide si i=0.

Formulation 3 : $P_3(i) = (i \in [1, n] \Rightarrow x \notin t[0..i-1])$

Cette formulation est correcte.

---

[1] Pour illustrer plaisamment cette difficulté, voici une petite histoire associée à Bertrand Russel (mathématicien et philosophe, 1872-1970, considéré comme l'un des fondateurs de la logique moderne).

Un certain philosophe fut très choqué quand Bertrand Russel lui apprit qu'une proposition fausse implique n'importe quelle proposition. Il dit à Russel : « Voulez-vous dire qu'il résulte de la proposition "deux plus deux font cinq" que vous êtes la Pape ? » et Russel répondit « Oui ». Le philosophe lui demanda alors : « Pouvez-vous le prouver ? ». Russel répondit : « Certainement », et inventa sur le champ la démonstration suivante :

1) Supposons que 2+2 = 5
2) On retranche 2 aux deux membres et on obtient 2 = 3
3) On transpose pour obtenir 3 = 2
4) Enfin on retranche 1 aux deux membres, ce qui donne 2 = 1.

A présent, le Pape est moi sommes 2. Mais 2 égale 1. Par conséquent le Pape et moi sommes 1. Donc je suis le Pape.

Deux propositions p et q sont équivalentes ssi $(p \Rightarrow q) \wedge (q \Rightarrow p)$. On notera $p \Leftrightarrow q$ ou bien encore $p = q$.

8.1.3 Quantificateur – Négation

Un prédicat est une formule logique qui contient des variables. Une variable soumise à un quantificateur est dite liée par ce quantificateur.

Dans un énoncé logique, l'ordre des quantificateurs $\forall$ et $\exists$ n'est pas indifférent.

Exemple

$\exists y \ \forall x \ P(x, y)$ signifie $\forall x \ P(x, a)$ où a est une constante

$\forall x \ \exists y \ P(x, y)$ signifie $\forall x \ P(x, f(x))$ où f(x) est une fonction de x

Donc $\exists y \ \forall x \ P(x, y) \Rightarrow \forall x \ \exists y \ P(x, y)$ mais la réciproque n'est pas vraie.

Les énoncés $\forall x \ \exists y \ P(x, y)$ et $\exists y \ \forall x \ P(x, y)$ ne sont donc pas équivalents.

Les principales règles relatives à la manipulation des quantificateurs et à la négation sont les suivantes :

- ordre des quantificateurs :
  - $\forall x \ \forall y \ P(x, y) \Leftrightarrow \forall y \ \forall x \ P(x, y)$
  - $\exists x \ \exists y \ P(x, y) \Leftrightarrow \exists y \ \exists x \ P(x, y)$
  - $\exists x \ \forall y \ P(x, y) \Rightarrow \forall y \ \exists x \ P(x, y)$ // noter la non équivalence
- factorisation des quantificateurs :
  - $\forall x \ P(x) \vee \forall x \ Q(x) \Rightarrow \forall x \ (P(x) \vee Q(x))$ // noter la non équivalence
  - $\forall x \ P(x) \wedge \forall x \ Q(x) \Leftrightarrow \forall x \ (P(x) \wedge Q(x))$
  - $\exists x \ P(x) \vee \exists x \ Q(x) \Leftrightarrow \exists x \ (P(x) \vee Q(x))$
  - $\exists x \ (P(x) \wedge Q(x)) \Rightarrow \exists x \ P(x) \wedge \exists x \ Q(x)$ // noter la non équivalence
- négation
  - $\neg (P \vee Q) \Leftrightarrow (\neg P \wedge \neg Q)$
  - $\neg (P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$
  - $\neg (P \Rightarrow Q) \Leftrightarrow (P \wedge \neg Q)$
  - $\neg (\forall x \ P) \Leftrightarrow \exists x \ (\neg P)$
  - $\neg (\exists x \ P) \Leftrightarrow \forall x \ (\neg P)$

Exemple

L'énoncé « Tous les éléments de A sont éléments de B » se traduit par : $\forall x \ (x \in A \Rightarrow x \in B)$. La négation de cet énoncé est : $\exists x \ (x \in A \wedge x \notin B)$, qui signifie : « Certains éléments de A ne sont pas des éléments de B ».

Exemple

La définition de O(f(x)) (cf section suivante) est :

$$\exists c \ \ \exists n_0 \ \ \text{tels que} \ \ \forall n \geq n_0 \ \ g(n) \leq c*f(n)$$

Elle signifie en fait :

$$\exists c \ \ \exists n_0 \ \ \forall n \ \ (n \geq n_0 \Rightarrow g(n) \leq c*f(n))$$

La négation de cet énoncé est :

$$\forall c \ \ \forall n_0 \ \ \exists n \ \ (n \geq n_0 \wedge g(n) > c*f(n))$$

8.2 Borne asymptotique d'une fonction

Soient f et g deux applications de N dans N.

Borne inférieure asymptotique. f(n) est une borne inférieure asymptotique de g(n) ssi, à partir d'une certaine valeur de n, f(n) est toujours inférieur à g(n) à un coefficient constant près. On dit alors que g domine asymptotiquement f et on note $g(n) = \Omega(f(n))$.

$$g(n) = \Omega(f(n)) \Leftrightarrow \exists\, c > 0 \quad \exists\, n_0 > 0 \quad \text{tels que} \quad \forall\, n \geq n_0 \quad c*f(n) \leq g(n)$$

Intuitivement, cela signifie que *g* croît au moins aussi vite que *f* quand n est grand.

Borne supérieure asymptotique. f(n) est une borne supérieure asymptotique de g(n) ssi, à partir d'une certaine valeur de n, f(n) est toujours supérieur à g(n) à un coefficient constant près. On dit alors que g est dominée asymptotiquement par f et on note $g(n) = O(f(n))$.

$$g(n) = O(f(n)) \Leftrightarrow \exists\, c > 0 \quad \exists\, n_0 > 0 \quad \text{tels que} \quad \forall\, n \geq n_0 \quad g(n) \leq c*f(n)$$

$$\Leftrightarrow f(n) = \Omega(g(n))$$

Intuitivement, cela signifie que *g* ne croît pas plus vite que *f* quand n est grand.

> *Nota. Ne pas confondre avec la notation o (petit o) : g(x) = o(f(x)) signifie que la fonction g est négligeable devant la fonction f quand n tend vers une valeur particulière. Par exemple : $(1 + x)^a = 1 + ax + o(x)$ quand $x \rightarrow 0$ (développement limité d'ordre 1 de $(1 + x)^a$ au voisinage de 0). La notation o est d'usage courant en mathématique mais plus rare en informatique.*

Borne approchée asymptotique. f(n) est une borne approchée asymptotique de g(n) ssi, à partir d'une certaine valeur de n, f(n) et g(n) sont similaires à un coefficient constant près. On dit alors que f et g sont asymptotiquement équivalents et on note $g(n) = \theta(f(n))$.

$$g(n) = \theta(f(n)) \Leftrightarrow \exists\, c > 0 \quad \exists\, d > 0 \quad \exists\, n_0 > 0 \quad \text{tels que } \forall\, n \geq n_0 \quad c*f(n) \leq g(n) \leq d*f(n)$$

$$\Leftrightarrow g(n) = O(f(n)) \text{ et } f(n) = O(g(n))$$

Intuitivement, cela signifie que g et f croissent aussi vite l'une que l'autre quand n est grand.

Remarques :

- En toute rigueur, O(f(n)) est l'ensemble des fonctions g(n) telles que :
  $\exists\, c > 0 \quad \exists\, n_0 > 0 \quad \text{tels que} \quad \forall\, n \geq n_0 \quad g(n) \leq c*f(n)$
  donc on devrait adopter une notation ensembliste : $g(n) \in O(f(n))$. La notation $g(n) = O(f(n))$ est néanmoins couramment utilisée, et c'est celle que nous adopterons. Ceci est également valable pour Ω et θ.
- La borne supérieure asymptotique fournie par la notation O peut-être ou non asymptotiquement approchée. Idem pour la borne inférieur asymptotique fournie par la notation Ω.
- En pratique, les termes d'ordre inférieur d'une fonction peuvent être ignorés lorsqu'on détermine les bornes asymptotiques, car ils ne sont pas significatifs pour n grand. En outre le coefficient du terme d'ordre supérieur peut être ignoré, puisque cela ne modifie les constantes c et d que d'un facteur constant égal à ce coefficient.

Exemple

Soient $f(n) = n^2 + n + 10$ et $g(n) = 100\, n + 5$

Pour n grand, $f(n) \approx n^2$ et $g(n) \approx 100\ n$. Pour n grand et à un coefficient constant près, f(n) se comporte comme n^2 et g(n) comme n.

D'où : $g(n) = O(f(n))$ $g(n) \neq \theta(f(n))$ $g(n) = \theta(n)$ $g(n) = O(n)$
$f(n) = O(n^2)$ $f(n) = \theta(n^2)$ $f(n) \neq \theta(g(n))$

Exemples

$n = O(n)$ $2n = O(3n)$ $n+2 = O(n)$ $n^{1/2} = O(n)$ $log(n) = O(n)$

8.3 Ensemble

On rappellera simplement quelques définitions et notations de base.

Ensemble, éléments. Un ensemble est une collection d'objets tous distincts appelés éléments de l'ensemble.

Un ensemble peut être défini *en extension* (par la liste de ses éléments) ou *en compréhension* (par la propriété caractérisant ses éléments).

Appartenance. Si x est élément d'un ensemble E, on notera : $x \in E$

Ensemble vide. L'ensemble vide est noté $\varnothing$ ou { }

Cardinal. Le cardinal d'un ensemble E, noté card(E) ou |E| ou #E, est le nombre d'éléments de cet ensemble.

Singleton. Un singleton est un ensemble d'un seul élément.

Inclusion. Les énoncés suivants expriment l'inclusion de deux ensembles et sont équivalents :

- l'ensemble A est inclus dans l'ensemble B
- A est un sous-ensemble de B
- A est une partie de B
- $A \subset B$ $(A \subseteq B \Leftrightarrow A \subset B \vee A = B)$
- $\forall x \in E,\ x \in A \Rightarrow x \in B$ (où E représente le référentiel)

Ensemble des parties d'un ensemble. L'ensemble des parties d'un ensemble E fini de n éléments est l'ensemble noté 2^E des 2^n sous-ensembles (ou parties) de E.

Exemple. Soit l'ensemble E = {a, b, c}. 2^E = { ∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a ,b,c} }

Opérations ensemblistes. Soit E un référentiel et A et B deux parties de E

- l'**union** de A et B est définie par : $A \cup B = \{ x \in E \mid x \in A \vee x \in B \}$
- l'**intersection** de A et B est définie par : $A \cap B = \{ x \in E \mid x \in A \wedge x \in B \}$
- la **différence** de A et B est définie par : $A \setminus B = \{ x \in E \mid x \in A \wedge x \notin B \}$
- le **complémentaire** de A est défini par : $\overline{A} = \{x \in E \mid x \in E \wedge x \notin A \}$
- la **fonction caractéristique** de A est l'application notée f_A définie par :

  $f_A : E \rightarrow \{0, 1\}$ telle que $f_A(x)=1$ si $x \in A$; $f_A(x)=0$ si $x \notin A$

8.4 Relation binaire – Relation d'ordre – Fermeture

On rappellera simplement quelques définitions et notations de base.

Produit cartésien. Le produit cartésien de deux ensembles A et B est défini par :

$A \times B = \{ (a, b) \mid x \in A \wedge x \in B \}$ où (a, b) dénote une paire ordonnée

Relation binaire. Une relation binaire (relation d'arité 2) est un sous-ensemble de $A \times B$. Une relation binaire sur un ensemble A est un sous-ensemble de $A \times A$

Soit une relation binaire $\Re$ sur un ensemble A :

- $\Re$ est **réflexive** ssi $\forall a \in A \ a\Re a$
- $\Re$ est **irréflexive** ssi aucun élément de A n'est en relation avec lui-même, c'est-à-dire $\forall a \in A \ a\not\Re a$
- $\Re$ est **symétrique** ssi $\forall a \in A \ \forall b \in A \ a\Re b \Rightarrow b\Re a$
- $\Re$ est **antisymétrique** ssi $\forall a \in A \ \forall b \in A \ a\Re b \wedge b\Re a \Rightarrow a=b$
- $\Re$ est **transitive** ssi $\forall a \in A \ \forall b \in A \ \forall c \in A \ a\Re b \wedge b\Re c \Rightarrow a\Re c$

Relation d'équivalence. Une relation binaire $\Re$ qui est réflexive, symétrique et transitive sur un ensemble A est appelée relation d'équivalence

Relation d'ordre (au sens large). Une relation binaire $\Re$ qui est réflexive, antisymétrique et transitive sur un ensemble A est appelée relation d'ordre (ou simplement ordre). On dit alors que $(A, \Re)$ est ordonné.

Eléments comparables. Si $\Re$ est une relation d'ordre sur A et si deux éléments a et b de A vérifient $a\Re b$ ou $b\Re a$, alors a et b sont dits comparables.

Relation d'ordre strict. Une relation binaire $\Re$ qui est irréflexive, antisymétrique et transitive (en toute rigueur irréflexif et transitif implique antisymétrique) sur un ensemble A est appelée relation d'ordre strict (ou simplement ordre strict). On dit alors que $(A, \Re)$ est strictement ordonné. Si deux éléments a et b de A vérifient $a\Re b$ ou $b\Re a$, ils sont dits strictement comparables.

Relation d'ordre total. Une relation d'ordre $\Re$ sur un ensemble A est dite relation d'ordre total (ou simplement ordre total) ssi deux éléments quelconques de A sont toujours comparables, c'est-à-dire :

$\forall a \in A \ \forall b \in A$, soit $a\Re b$ soit $b\Re a$

On dit alors que $(A, \Re)$ est totalement ordonné.

Exemple : $\leq$ est un ordre total sur Z, R ou Q (mais pas sur C).

Exemple : $<$ est un ordre total strict sur Z, R ou Q (mais pas sur C).

Exemple.

Soit $(E, <)$ un ensemble totalement ordonné ; soit $E^2 = E \times E$. La relation d'ordre « précède » définie sur l'ensemble E^2 par :

$$(a_1, a_2) \text{ « précède » } (b_1, b_2) \Leftrightarrow (a_1 < b_1) \vee ((a_1 = b_1) \wedge (a_2 < b_2))$$

*est un ordre total strict. Cette relation est l'**ordre lexicographique** sur E^2.*

Relation d'ordre partiel. Une relation d'ordre pour laquelle il existe des éléments incomparables est dite relation d'ordre partiel (ou simplement ordre partiel) ; on dit alors que $(A, \Re)$ est partiellement ordonné.

Exemple : dans N, la relation « est divisible par » est une relation d'ordre partiel.

Exemple : dans une famille, la relation « est aïeul de » est une relation d'ordre partiel strict.

Exemple : dans le plan, la relation « est plus près de l'origine que » est une relation d'ordre partiel strict.

Fermeture transitive. Soit une relation binaire $\Re$ sur un ensemble A. On appelle fermeture transitive de $\Re$ la relation binaire $\Re^+$ définie sur A par :

$$\Re^+ = \bigcup_{i \geq 1} \Re^i$$

où $\Re^i$ est définie par récurrence sur $i \in N$:

$$\Re^0 = \{ (a, a) \mid a \in A \}$$

$$\Re^{i+1} = \{ (a, c) \in A \times A \mid \exists b \in A \ a \Re^i b \wedge b \Re c \}$$

Fermeture réflexive transitive. Soit une relation binaire $\Re$ sur un ensemble A. On appelle fermeture transitive de $\Re$ la relation binaire $\Re^*$ définie sur A par :

$$\Re^* = \bigcup_{i \geq 0} \Re^i = \Re^+ \cup \{ (a, a) \mid a \in A \}$$

Exemple. Considérons la relation « a gagné contre » définie sur l'ensemble A des joueurs d'un tournoi. La relation « est plus fort que » sur ce même ensemble de joueurs se définit par : $\forall a \in A \ \forall b \in A$ a « est plus fort que » b $\Leftrightarrow$ (a « a gagné contre » b) $\vee$ ($\exists a_1 \in A$ a « a gagné contre » $a_1 \wedge a_1$ « a gagné contre » b) $\vee$ ($\exists a_1 \in A \ \exists a_2 \in A$ …) … . Donc « est plus fort que » est la fermeture transitive de « a gagné contre ».

8.5 Distance

Une distance (ou métrique) sur un ensemble E est une application $d : E \times E \rightarrow R^+$ telle que :

- $\forall x \in E \ \forall y \in E \ d(x, y) = d(y, x)$ Symétrie
- $\forall x \in E \ \forall y \in E \ d(x, y) = 0 \Leftrightarrow x = y$
- $\forall x \in E \ \forall y \in E \ \forall z \in E \ d(x, z) \leq d(x, y) + d(y, z)$ Inégalité triangulaire

9 ANNEXE B – Rappels de programmation

9.1 Conception d'une structure de données

Une structure de données est un agrégat (collection structurée) de cellules mémoire. Chaque « cellule » est un bloc de mémoire ayant la taille appropriée pour contenir l'objet du type élémentaire ou complexe donné.

La conception d'une structure de données se réalise, *en assurant une cohérence descendante*, en trois étapes :

1) spécification fonctionnelle : il s'agit de décrire les fonctionnalités souhaitées ; c'est le stade du quoi, du fonctionnel
2) spécification logique : il s'agit de spécifier les grands choix d'organisation de la structure de données afin de permettre une implémentation efficace des fonctionnalités
3) spécification physique : il s'agit de définir concrètement tous les types et sous-programmes antérieurement spécifiés ; c'est le stade du comment, du structurel.

Une même description fonctionnelle peut conduire à plusieurs spécifications logiques possibles ; une même spécification logique peut conduire elle-même à plusieurs représentations physiques possibles.

9.1.1 Spécification fonctionnelle

Il s'agit de décrire rigoureusement toutes les fonctionnalités souhaitées, avec notamment leurs pré-conditions et comportements respectifs, en faisant abstraction de l'implémentation. Par exemple par des énoncés E {P} S.

Les grandes catégories de fonctionnalités sont typiquement :

- les constructeurs
- les fonctions d'accès et les observateurs
- les modificateurs

Exemple

Soit à réaliser un tampon de communication entre un processus émetteur et un processus récepteur qui fonctionnent en asynchronisme à des débits différents. Le tampon sera une file d'attente – c'est-à-dire une liste d'attente à discipline « premier arrivé = premier servi » – à capacité non bornée. On ne se posera pas ici le problème de la synchronisation de l'accès à la structure de données partagée qu'est ce tampon

On notera $\langle e_k\ e_{k-1} \ldots e_2\ e_1 \rangle$ *une file d'attente de k éléments, où* e_1 *représente le premier élément arrivé dans la file d'attente et identifie donc le début de la file, et* e_k *représente le dernier élément arrivé dans la file d'attente et identifie donc la fin de la file.*

Opérations souhaitées :

Créer une file d'attente vide	:	*file ← créer()*
Ajouter un élément à la file d'attente	:	*ajouter(file, élément)*

Retirer le 1er élément de la file d'attente : élément ← retirer(file)

Longueur de la file d'attente : n ← longueur(file)

Propriétés exprimées dans le formalisme E {P} S :

( ) { file ← créer( ) } (file = ⟨ ⟩)

(file=⟨e_k … e_2 e_1⟩ ∧ k≥0) { ajouter(file, e) } (file=⟨e e_k … e_2 e_1⟩)

(file=⟨e_k … e_2 e_1⟩ ∧ k≥1) { e ← retirer(file) } (file=⟨e_k … e_2⟩ ∧ e=e_1)

(file=⟨e_k … e_2 e_1⟩ ∧ k≥0) { n ← longueur(file) } (n=cardinal(⟨e_k … e_2 e_1⟩))

9.1.2 Spécification logique

Cette étape définit les grands choix d'organisation de la structure de données en la décomposant en sous-ensembles plus simples qu'on sait représenter et de façon à assurer la faisabilité et l'efficacité des fonctionnalités souhaitées.

La spécification logique fournit une décomposition des objets répondant à la définition fonctionnelle en objets plus élémentaires, et une décomposition des opérations associées en opérations plus élémentaires.

Une même spécification fonctionnelle peut correspondre à plusieurs descriptions logiques possibles.

La distinction d'avec la phase de spécification physique peut être parfois subtile.

Il existe quatre grandes catégories de base d'organisation des données :

1) une organisation directe par le rang

   Quel que soit l'élément appartenant à la structure de données, l'élément est repérable par son indice ou son nom.

   L'accès direct à un élément quelconque se réalise à temps constant.

2) une organisation séquentielle

   Ce sont les listes, et leurs différents cas particuliers : piles (listes d'attente à discipline « dernier arrivé = premier sorti »), files d'attente (listes d'attente à discipline « premier arrivé = premier sorti »), … .

   Une telle organisation de données suppose a minima : une porte d'accès (par exemple au « premier » élément), offerte par un opérateur particulier ; l'accès à un élément quelconque autre ne peut se faire que via le « prédécesseur » de cet élément.

   C'est le passage d'un élément à un autre qui se réalise à temps constant.

   Intérêt de cette organisation : facilité d'insertion / suppression d'un élément.

3) une organisation arborescente

4) une organisation relationnelle en graphes

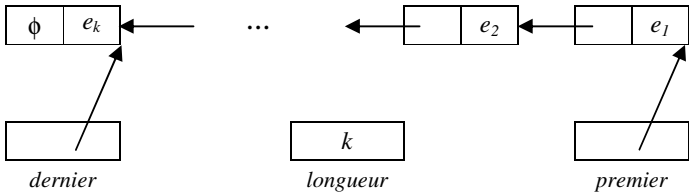
L'organisation logique retenue peut être une combinaison de tout ou partie de ces organisations de base.

Exemple

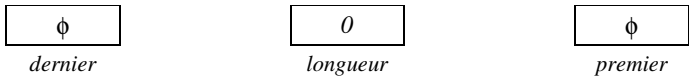
Reprenons l'exemple de la file d'attente de la section précédente.

Comme c'est une file d'attente ordinaire, sans gestion de priorité, le choix d'une simple organisation séquentielle est approprié. Il reste à décider le sens dans lequel

chaîner les éléments : des chaînages e_i vers e_{i+1} ou l'inverse ? Pour chaque opération à réaliser, on évalue quelle organisation est la meilleure quant à la faisabilité et l'efficacité de l'opération. Si besoin, on crée des variables complémentaires pour atteindre le but recherché. Cette analyse conduit à l'organisation suivante :



Une file d'attente vide sera :



Avec cette organisation, les quatre opérations de base pourront s'exécuter en $\theta(1)$.

9.1.3 Spécification physique

Il s'agit ici de choisir une incarnation concrète finale : choix d'une structure de données, écriture d'un sous-programme pour chaque fonctionnalité souhaitée.

Une même spécification logique peut correspondre à plusieurs représentations logiques possibles.

Une structure de données est un agrégat de cellules mémoire. Les langages de programmation offrent deux mécanismes d'agrégation (non exclusifs) :

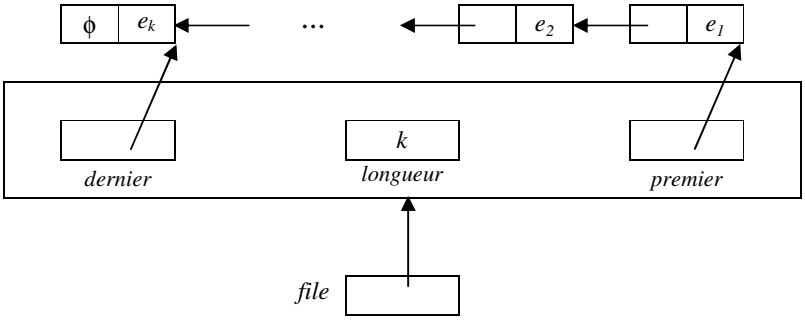
- par contiguïté : tableau, structure au sens C, …
- par chaînage : à chaque cellule est associé un champ particulier qui réalise un lien vers une autre cellule. Ce lien peut être une adresse physique (un pointeur) ou une adresse logique (un indice). Le chaînage permet de rendre plus efficace l'ajout et l'insertion d'un élément au sein d'une structure de données dont l'organisation physique se veut refléter une relation d'ordre.

Exemple

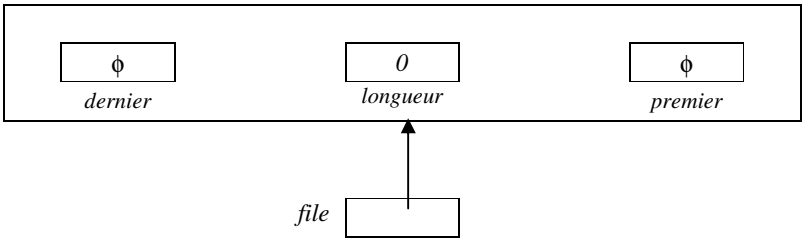
Reprenons l'exemple de la file d'attente de la section précédente.

S'il avait été recherché une file d'attente à « capacité bornée », il aurait été possible de l'implémenter efficacement par un tableau géré de façon circulaire : les éléments en file auraient alors constitué une séquence d'éléments du tableau ; les variables dernier et premier auraient été des indices.

Dans notre cas, la spécification « capacité non bornée » induit la mise en œuvre de structures dynamiques chaînées. Par ailleurs, les variables de gestion mises en œuvre (premier, longueur et dernier) n'ont de sens qu'associées : nous allons donc les regrouper dans une structure. D'où la structure de données finale :



Et pour une file vide :



L'écriture en C des types et des sous-programmes de gestion est fournie en annexe 9.3.2

9.2 Allocation dynamique

Les variables allouées dynamiquement lors de l'exécution sont stockées dans un espace dédié de la mémoire et de portée globale.

Exemple d'allocation / libération dynamique d'un tableau t[0..n-1] d'éléments de type Elmt.

	En C	**En C++**	**En Java**
Déclaration	`Elmt * t ;`	`Elmt * t`	`Elmt[] t ;`
Allocation dynamique	`t = (Elmt*)malloc( n*sizeof(Elmt));`	`t = new Elmt[n] ;`	`t = new Elmt[n];`
Libération mémoire	`free(t) ;`	`delete[] t ;`	`t = null ;`

Exemple d'allocation / libération dynamique d'une structure de données :

	En C	**En C++**	**En Java**

Définition des types	`typedef struct {`
	`  int  clef ;`
	`  char* nom ;`
	`} Elmt ;`
	
	`typedef Elmt* Element;`
		`typedef struct {`
		`  int  clef ;`
		`  char* nom ;`
		`} Elmt ;`
		
		`typedef Elmt* Element;`
			`public class Element {`
			`  int clef ;`
			`  String nom ;`
			`  public Element(int k,`
			`  String s){`
			`  clef = k ;`
			`  nom = s ;`
			`  }`
			`  ...`
			`} ;`

	En C	En C++	En Java
Déclaration	`Element p ;`	`Element p ;`	`Element p ;`
Allocation dynamique	`p = (Element)malloc( sizeof(Elmt));`	`p = new Element ;`	`P = new Element(15, "abc");`
Initialisation	`p->clef = 15 ;` `p->nom = "abc" ;`	`p->clef = 15 ;` `p->nom = "abc" ;`	
Libération mémoire	`free(p) ;`	`delete p ;`	`P = null ;`

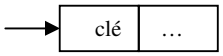
9.3 Chaînage

9.3.1 Chaînage dans un tableau

Ce sont les indices qui servent de lien. Dans ce contexte, un lien est parfois appelé curseur.

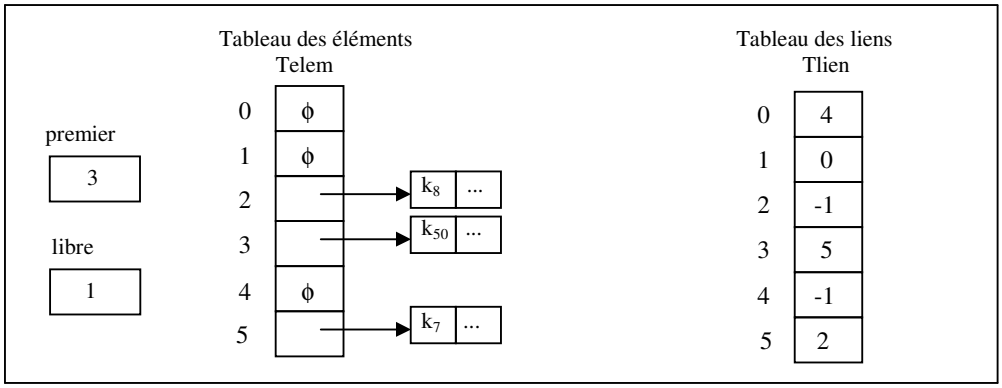
Exemple

Soit une liste d'éléments, chaque élément étant composé d'une clé et de données satellites. On représentera graphiquement un élément de la façon suivante :



le champ de droite « … » représentant les données satellites.

La liste des éléments successifs de clé k_{50}, k_7 et k_8 peut être implémentée, par exemple, par la structure de données suivante :



Cette organisation comporte un double chaînage : le chaînage des cellules pleines (c.-à-d. des éléments), d'une part, et le chaînage des cellules vides, d'autre part.

`premier` *est l'indice du premier élément de la liste d'éléments. Les autres éléments ont pour indices successifs :* `Tlien[premier]` *et* `Tlien[Tlien[premier]]`*. Il n'y a pas d'autre élément car* `Tlien[Tlien[Tlien[premier]]=-1`*.*

`libre` *est l'indice de la « première » cellule de la liste des cellules vides. Les autres cellules libres ont pour indices successifs :* `Tlien[libre]` *et* `Tlien[Tlien[libre]]`*. Il n'y a pas d'autre cellule libre car* `Tlien[Tlien[Tlien[libre]]=-1`*.*

Ainsi, liste vide impliquerait `premier=-1` *; une liste pleine impliquerait* `libre=-1`*.*

Une telle liste d'éléments se gère ensuite séquentiellement.

9.3.2 Chaînage de structures dynamiques

Exemple

Programmation en C-ANSI de la file d'attente définie au chapitre 9.1.3.

```
typedef struct Cell {
      Element element ;
      struct Cell * suivant ;
} Cell ;

typedef Cell * PtrCell ;

typedef struct {
      PtrCell premier ;
      PtrCell dernier ;
      int     longueur ;
} SFile, *File ;

File creer() {
      File f = (File)(malloc(sizeof(SFile))) ;
      f->premier = NULL ;
      f->dernier = NULL ;
      f->long    = 0 ;
      return f ;
}

void ajouter (File f, Element e) {
      PtrCell p = (PtrCell)(malloc(sizeof(Cell))) ;
      p->element = e ;
      p->suivant = NULL ;
      PtrCell q = f->dernier ;
      if ( q==NULL ) {                /* cas liste vide */
            f->premier = p ;
      } else {                        /* cas liste non vide */
            q->suivant = p ;
      }
      f->dernier = p ;
      f->longueur++ ;
}

Element retirer(File f) {
      if ( (f->longueur != 0 ) {      /* cas liste non vide */
            PtrCell p = f->premier ;
            Element e = p->element ;
            f->premier = p->suivant ;
            free(p) ;
            f->longueur-- ;
            return e ;
      } else erreur() ;
}
```

10 ANNEXE C – CALCUL DE COMPLEXITE ET RÉCURRENCE

Lorsque le temps de calcul d'un algorithme en fonction de la taille du problème à traiter est exprimé par une relation de récurrence (situation naturelle pour les algorithmes récursifs), comment en déduire la complexité de cet algorithme ? La détermination rigoureuse de la complexité algorithmique n'est pas toujours chose aisée. Il est proposé ici quatre approches pratiques courantes [3] :

- Méthode par développement itératif de la récurrence
- Méthode par détermination de l'arbre des coûts
- Méthode par substitution et récurrence mathématique
- Méthode générale pour des récurrences de la forme T(n) = a T(n/b) + f(n)

10.1 Méthode par développement itératif

Principe : itérer la récurrence par valeurs croissantes, exprimer chaque résultat par une somme de termes fonctions de la taille du problème et des conditions initiales, et dégager progressivement une forme générale.

Cette méthode nécessite des manipulations algébriques pour une mise en forme appropriée des résultats. Elle permet de dégager une hypothèse de solution qu'il reste à vérifier par récurrence mathématique[1].

En pratique, peu ou prou, deux formules basiques sont souvent utiles :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=0}^{p} 2^i = 2^{p+1} - 1$$

Exemple.

Soit un algorithme dont le temps d'exécution s'exprime par la récurrence :

$$T(n=1) = a$$
$$T(n=2^{p>0}) = b + c\,n + 2\,T(n/2)$$

$$T(n=2=2^{p=1}) = b + 2\,c + 2\,a$$

[1] Pour illustrer plaisamment la difficulté à conduire des raisonnements logiques corrects, un peu de détente avec le paradoxe de l'interrogation surprise (D. J. O'Connor, "Pragmatic Paradoxes", *Mind* 1948, Vol. 57, p 358).

Un professeur annonce à ses élèves : « Il y aura une interrogation surprise la semaine prochaine ». Par là, il veut dire précisément trois choses : une interrogation aura lieu durant un cours soit le lundi, soit le mardi, soit le mercredi, soit le jeudi, soit le vendredi ; juste avant le début de l'interrogation, l'élève ne pourra avoir la certitude que l'interrogation va avoir lieu ; une unique interrogation aura lieu.

Un élève futé fait le raisonnement suivant : « Si jeudi soir, l'interrogation n'a pas eu lieu, alors je serai certain qu'elle est pour vendredi. Ce ne sera donc plus une surprise. L'interrogation ne peut donc avoir lieu vendredi parce c'est le dernier jour possible. Mais puisque l'interrogation ne peut avoir lieu le dernier jour, l'avant-dernier jour devient de facto, le dernier jour possible. Ainsi, par récurrence, on en déduit que l'interrogation ne peut pas avoir lieu ! ».

Où est le problème ?

$T(n=4=2^{p=2}) = b + 4\,c + 2\,T(2) = b(1+2) + 2\times 4c + 4\,a$

$T(n=8=2^{p=3}) = b + 8\,c + 2\,T(4) = b(1+2+4) + 3\times 8c + 8\,a$

…

$$T(n=2^{p>0}) = b\sum_{i=0}^{p-1} 2^i + p\,n\,c + n\,a$$

$$= b\,(2^p - 1) + p\,n\,c + n\,a$$

$$= b\,(n-1) + lg(n)\,n\,c + n\,a$$

$$= \theta(n\,log(n))$$

10.2 Méthode par détermination de l'arbre des coûts

Principe : développer la récurrence en un arbre dont les nœuds représentent les coûts à chaque étape, puis, par niveau d'abord puis globalement pour tout l'arbre, sommer et borner les coûts.

Cette méthode s'avère pratique, notamment, quand la récurrence décrit le temps d'exécution d'un algorithme de type « *diviser pour régner* ».

Exemple.

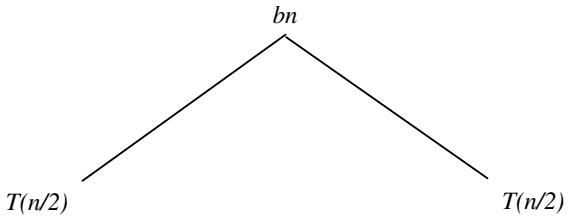
Soit un algorithme dont le temps d'exécution s'exprime par la récurrence :

$T(n=1) = a$

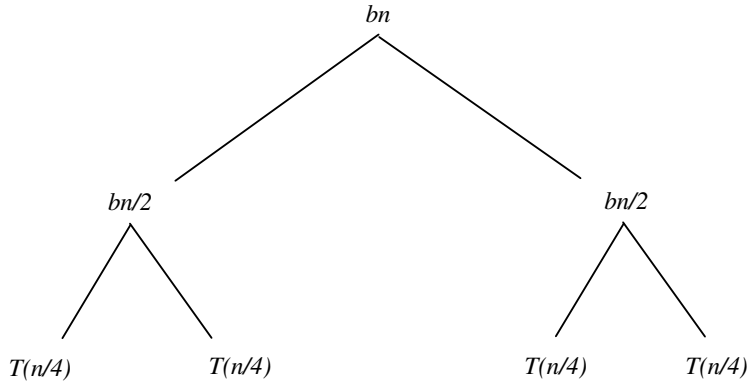
$T(n>1) = 2\,T(n/2) + bn$

Construisons l'arbre des coûts.

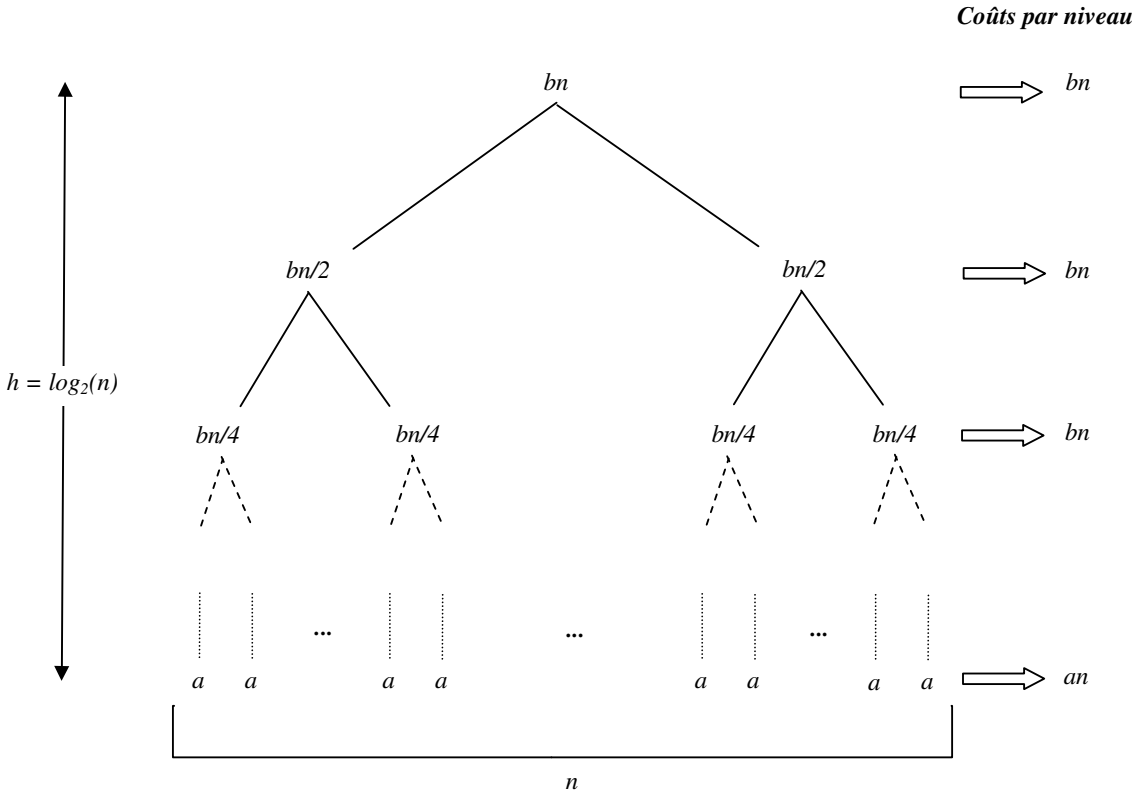
T(n) se développe progressivement en :



puis



puis ..., puis in fine :



Au total, $T(n) = b\,n\,log_2(n) + a\,n = \theta(n\,log(n))$

10.3 Méthode par substitution et récurrence mathématique

Principe : pressentir la forme de la solution (en O ou θ ou Ω), substituer cette forme aux appels réalisés dans la relation de récurrence, et utiliser une induction mathématique pour prouver que le résultat est correct.

Cette méthode n'est utilisable que si la forme de la réponse est assez facile à deviner. Elle présente toutefois un grand intérêt car elle est puissante et peut permettre de borner une récurrence par excès ou par défaut (utile en particulier quand des parties entières ($\lfloor n/b \rfloor$ ou/et $\lceil n/b \rceil$) sont utilisées dans la relation de récurrence).

Exemple.

Le temps d'exécution d'un parcours infixe d'arbre s'exprime par la récurrence :

$T(n=0) = a$

$T(n>0) = b + T(n1) + T(n-n1-1)$ avec $0 \leq n1 < n$

L'algorithme passant par chacun des n nœuds de l'arbre en y effectuant un traitement en $\theta(1)$, il est plus que raisonnable de supposer que $T(n) = \theta(n)$. Supposons donc que $T(n) = c + d\,n$, et vérifions cette hypothèse par récurrence mathématique.

$T(n=0) = c$. Mais $T(n=0) = a$. Donc $c = a$.

$T(n=1) = c + d$. Mais $T(n=1) = b + 2a$. Donc $d = a + b$.

L'hypothèse de récurrence est donc : $T(n>1) = a + (a+b)n$. Supposons cette hypothèse vraie jusque n-1 et vérifions la pour n.

Par énoncé : $T(n>1) = b + T(n1) + T(n-n1-1)$ avec $0 \leq n1 < n$

Comme $n1 < n$: $T(n1) = a + (a+b)n1$

Comme $n-n1-1 < n$: $T(n-n1-1) = a + (a+b)(n-n1-1)$

D'où, par substitution : $T(n>1) = b + a + (a+b)n1 + a + (a+b)(n-n1-1)$

Donc, pour tout n : $T(n) = a + (a+b)n$. *CQFD.*

Exemple.

Soit un algorithme dont le temps d'exécution s'exprime par la récurrence :

$T(n=1) = a$

$T(n>1) = 2\,T(\lfloor n/2 \rfloor) + n$

Faisons l'hypothèse que la solution est $T(n) = O(n \log_2(n))$, *c'est-à-dire que* $T(n) \leq c\,n \log_2(n)$ *pour une certaine constante* $c>0$, *et vérifions cette hypothèse par récurrence mathématique.*

Supposons cette hypothèse vraie pour $\lfloor n/2 \rfloor$, *c'est-à-dire que*
$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor)$

En substituant dans la récurrence on obtient :

$$\begin{aligned} T(n) &\leq 2\,c \lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor) + n \\ &\leq c\,n \log_2(n/2) + n \\ &= c\,n \log_2(n) - c\,n + n \\ &\leq c\,n \log_2(n) \quad \text{dès que } c \geq 1 \end{aligned}$$

Il reste à prouver que l'hypothèse est vraie initialement quand s'amorce l'induction :

$$\begin{aligned} T(n=2) &= 2\,a + 2 \\ &\leq c\,n \log_2(n) \quad \text{dès que } c \geq a+1 \\ T(n=3) &= 2\,a + 3 \\ &\leq c\,n \log_2(n) \quad \text{dès que } c \geq a+1 \end{aligned}$$

CQFD.

10.4 Méthode générale quand T(n) = a T(n/b) + f(n)

Le temps d'exécution d'un algorithme de type « *diviser pour régner* » divisant le problème de taille `n` en `a` sous-problèmes de taille `n/b` se définit par une relation de récurrence de la forme :

$T(n \leq c) = \theta(1)$

$T(n > c) = D(n) + a\,T(n / b) + C(n)$

où `D(n)` est le temps de division du problème en sous-problèmes, `a T(n/b)` le temps de résolution des `a` sous-problèmes, et `C(n)` le temps de combinaison des sous-problèmes résolus et de construction de la solution finale.

Le théorème suivant est une recette pour résoudre de telles récurrences, c'est-à-dire des récurrences de la forme $T(n) = a\,T(n / b) + f(n)$, sous certaines conditions.

Théorème [3]

Soient $a \geq 1$ et $b > 1$ deux constantes, $f(n)$ une fonction asymptotiquement positive, et $T(n)$ définie pour les entiers naturels par la récurrence $T(n) = a\,T(n / b) + f(n)$ où n/b peut signifier aussi bien $\lfloor n/b \rfloor$ que $\lceil n/b \rceil$.

$T(n)$ peut alors être bornée asymptotiquement de la façon suivante :

1) Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \theta(n^{\log_b a})$
2) Si $f(n) = \theta(n^{\log_b a})$, alors $T(n) = \theta(n^{\log_b a} \lg(n))$
3) Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $a\,f(n/b) \leq c\,f(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = \theta(f(n))$

Il est à noter que ces trois cas, pour lesquels il est possible de conclure de façon générale, ne recouvrent pas toutes les possibilités pour $f(n)$! Par exemple, ce théorème ne permet pas de conclure dans le cas $T(n) = 2\,T(n/2) + n \lg(n)$ car cette récurrence ne vérifie aucune des conditions.

Exemple d'application :

$T(n \leq c) = \theta(1)$ $T(n>c) = a\,T(n / b) + d\,n^k$	$a > b^k$	$\theta(n^{\log_b a})$
	$a = b^k$	$\theta(n^k \lg(n))$
	$a < b^k$	$\theta(n^k)$

11 ANNEXE D – DÉCIDABILITÉ – NP-COMPLÉTUDE

11.1 Décidabilité

Le terme de décidabilité réfère à la fois à un concept de la logique mathématique et à un concept de l'algorithmique. Ces deux concepts, quoique distincts, sont liés. Dans les deux cas, il s'agit d'exprimer l'idée qu'on ne peut pas toujours conclure.

Décidabilité logique.

Une assertion est une affirmation (ex : la somme de deux nombres impairs est paire). Une assertion est consistante si elle ne peut pas être à la fois vraie et fausse (par exemple, l'assertion « Cette phrase est fausse » est inconsistante).

Une assertion est dite décidable dans une théorie axiomatique si on peut la démontrer ou démontrer sa négation dans le cadre de cette théorie. Sinon, l'assertion est dite indécidable dans cette théorie.

Une théorie mathématique dans laquelle toute assertion est décidable est dite complète. Le théorème d'incomplétude de Gödel[1] assure que tout système axiomatique indépendant, cohérent et assez puissant pour décrire les nombres naturels est incomplet. C'est le cas, par exemple de l'arithmétique.

Décidabilité algorithmique.

Un problème de décision est dit décidable s'il existe un algorithme, c'est-à-dire une procédure mécanique qui termine en temps fini d'étapes, qui réponde par oui ou par non à la question posée par le problème.

Exemple de problème indécidable.

Le problème de l'arrêt d'un programme informatique est indécidable (l'indécidabilité a été prouvée par Turing[2]) ou trivial (réponse toujours vraie ou toujours fausse). Plus généralement, c'est le cas pour toute question sur les programmes informatiques qui ne dépend que du résultat du calcul (théorème de Rice).

11.2 Réduction polynomiale et NP-complétude

Il s'agit ici de définir et de justifier la NP-complétude d'un problème de façon un peu plus formelle qu'au chapitre 2.

---

[1] Kurt GÖDEL (1906 -1978), mathématicien et logicien autrichien, est renommé pour ses théorèmes en logique mathématique. En particulier son théorème d'incomplétude (qu'il a publié à 25 ans) : quel que soit le système axiomatique indépendant et assez puissant pour décrire les nombres naturels, si le système est cohérent, alors la cohérence des axiomes ne peut pas être prouvée au sein même du système. Ses théorèmes mirent fin à des siècles de tentatives de proposer un jeu d'axiomes définitif pour situer l'ensemble des mathématiques sur une base axiomatique.

[2] Alan TURING (1912-1954), mathématicien britannique, est un des pères fondateurs de l'informatique. Il a notamment été à l'origine de la formalisation des concepts d'algorithme et de calculabilité.

Définition. Dans le cadre des problèmes décidables, une réduction permet de ramener la décidabilité d'un problème à celle d'un autre problème. On dit qu'un problème P1 est réductible à un problème P2 s'il existe un algorithme résolvant P1 qui utilise un algorithme résolvant P2. Si l'algorithme résolvant P1 est polynomial *sous hypothèse* de considérer l'algorithme P2 de complexité constante (c.-à-d. en O(1)), alors on parle de **réduction polynomiale** et on dit que P1 est polynomialement réductible à P2.

Définition. Un problème est dit **NP-difficile** si tout problème de NP lui est polynomialement réductible.

Définition. Un problème est dit **NP-Complet** s'il est dans NP et s'il est NP-difficile.

Exemple référent de problème NP-Complet.

Considérons le problème SAT de la satisfiabilité d'une formule logique F de n variables x_1, .., x_n. Il s'agit de déterminer s'il existe une façon d'assigner les valeurs « vrai » ou « faux » aux variables afin de rendre l'expression F vraie. Voici un algorithme non-déterministe résolvant ce problème :

Pour i variant de 1 à n faire $x_i \leftarrow$ CHOIX(vrai, faux) FinPour ;
Si $F(x_1, .., x_n)$ est vraie alors F est satisfiable sinon F est non satisfiable.

où CHOIX est un oracle solutionnant en O(1) le problème difficile de déterminer la valeur vrai ou faux pour la variable considérée. Cet algorithme non-déterministe est clairement en $\theta(n)$ et donc ce problème est dans NP. Est-il pour autant NP-Complet ? Oui, car tout problème de NP peut se réduire polynomialement à SAT. La preuve originelle a été apportée par Cook[1] en 1971.

Exemple de preuve de NP-complétude par réduction polynomiale.

A delà du problème que nous allons considérer, cet exemple illustre la méthode générale de détermination de la NP-complétude d'un problème par réduction polynomiale.

Considérons le problème 3-SAT de la satisfiabilité d'une expression logique sous forme normale conjonctive comportant exactement 3 variables par clause (une telle expression est de la forme : $E_1 \wedge \ldots \wedge E_k$ où les E_i sont de la forme $x \vee y \vee z$). Le problème 3-SAT est-il NP-Complet ? 3-SAT est clairement dans NP, car SAT y est. Si l'on peut réduire polynomialement SAT à 3-SAT, alors, SAT étant NP-Complet et la réduction polynomiale étant transitive, 3-SAT sera prouvé NP-Complet. Soit donc à prouver que SAT se réduit polynomialement à 3-SAT. Voici les grandes étapes de la démonstration.

Etape1. On montre facilement que toute expression logique peut être mise sous une forme normale conjonctive équivalente du point de vue de la satisfiabilité, et cela en temps polynomial.

Etape2. On montre que toute expression logique sous forme normale conjonctive peut être mise sous une forme normale conjonctive comportant exactement trois variables par clause (nota : il faut introduire des variables supplémentaires) et équivalente du point de vue de la satisfiabilité, cela en temps polynomial.

[1] Stephen COOK (1939-) est un informaticien américain. Il a formalisé la notion de NP-complétude.

Etape3. Supposons maintenant qu'on dispose d'un algorithme A résolvant 3-SAT. On peut alors développer un algorithme B résolvant SAT en faisant appel à A :

Transformer $F(x_1, .., x_n)$ sous forme normale conjonctive F' ;

Transformer $F'(x_1, .., x_n)$ sous forme normale conjonctive comportant trois variables par clause, $G(x_1, .., x_n, y_1, .., y_m)$;

Déterminer si G est satisfiable en appliquant A à G ;

Si G est satisfiable, alors F l'est aussi pour les mêmes valeurs de $x_1, .., x_n$; sinon, F ne l'est pas non plus.

Sous hypothèse de considérer A de complexité O(1), l'algorithme B est polynomial et donc le problème SAT est polynomialement réductible au problème 3-SAT. En conséquence, SAT étant NP-Complet, 3-SAT est lui aussi NP-Complet

D'un point de vue pratique, prouver qu'un problème est NP-Complet rend utopique la recherche d'un algorithme polynomial le résolvant car cet algorithme résoudrait alors du même coup des centaines d'autres problèmes qui ont fait l'objet de recherches intensives … !

12 BIBLIOGRAPHIE ET SITES DE REFERENCE

Principales références bibliographiques :

[1] AHO Alfred, ULLMAN Jeffrey. *Concepts fondamentaux de l'informatique*. Dunod, 1993, 856 p.
ISBN : 2100031279

[2] COOK Stephen. « The complexity of theorem proving procedures ». *Proceedings of the third annual ACM symposium on theory of computing*, 1971, p. 151-152.

[3] CORMEN Thomas H., et al. *Introduction à l'algorithmique (2e éd.)*. Dunod, 2002, 1176 p.
ISBN : 2100039229

[4] CROCHEMORE Maxime, RYTTER Wojciech. *Text Algorithms*. Oxford University Press, 1994, 412 p.
ISBN : 019508090

[5] HOARE C. A. R. « An axiomatic basis for computer programming ». *Communication of the ACM,* vol. 12, 1969, p. 576-585.
www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf

[6] KNUTH Donald E. *The art of computer programming*, vol. 1-3. Addison Wesley, 1998, 896 p.
ISBN : 0201485419

[7] NATOWICZ René. *Cours d'algorithmique*. ESIEE I2.

[8] SEDGEWICK Robert, SCHIDLOWSKY Michael. *Algorithmes en Java (3e éd.)*. Pearson Education, 2002, 772 p.
ISBN : 2744070246

[9] WIKIPEDIA [encyclopédie en ligne]. www.wikipedia.fr

13 INDEX