Transparents

INFOB233

Programmation 2ème partie

http ://www.info.fundp.ac.be/~pys/cours/infob233/

Pierre-Yves SCHOBBENS

pys@info.fundp.ac.be

bureau 409

081/724990

24 décembre 2012

Table des matières

Références

– T. Cormen, C. Leiserson, R. Rivest, C. Stein *Introduction à l'algorithmique, 2ème éd.*, Dunod, Paris, 2002 (ISBN 2-10-003922-9).
pages 1-70, 121-153, 195-287, 315-382, 425-442.
Vous pouvez aussi utiliser l'ancienne édition :
T. Cormen, C. Leiserson, R. Rivest, *Introduction à l'algorithmique*, Dunod, Paris, 1994 (ISBN 2-10-003128-7).
– P. Berlioux et Ph. Bizard : Algorithmique : construction, preuve et évaluation de programmes de (ed. Dunod)

Ce cours (et le livre) présupposent la connaissance :

1. du langage de programmation Pascal ; Si vous voulez vous mettre à niveau, vous pouvez lire :
   (a) N. Wirth, K. Jensen. Pascal User Manual and Report, Springer Verlag
   (b) http ://www-ipst.u-strasbg.fr/pat/program/pascal.htm
2. Pour les exercices, du Langage de programmation C
3. des preuves mathématiques par induction
4. la preuve de programmes par pre- et post-conditions, par invariants de boucle. Vous pouvez lire :

   D. Gries, The Science of Programming, Springer

   R. Backhouse, Construction et vérification de programmes, Masson, Paris, 1989

   W. Vanhoof, Syllabus Méthodes de Programmation (1) (disponible sur WebCampus)

CHAPITRE 1

LA RÉCURSION

Une définition est **(directement) récursive** si la terme défini apparaît lui-même dans la définition

1.1 Récursion bien fondée

Le plus souvent, de telles “définitions” sont circulaires : elles ne définissent rien du tout.

Pour qu’une récursion définisse vraiment quelque chose, il faut qu’elle soit **bien fondée** : en l’utilisant on se ramène à des cas plus simples.

Exemple : `factorielle:` $\mathbb{N} \to \mathbb{N}$

```
factorielle(0) = 1
factorielle(n) = n * factorielle(n-1)  si n>0
```

1.1.1 Relation bien fondée

Une relation binaire quelconque $<$ “plus simple” est **bien fondée** ssi il ne peut y avoir de suite infinie strictement décroissante : $a_1 > a_2 > a_3 > \ldots$.

Ceci dépend de l’ensemble dans lequel on travaille :
par exemple, $<$ sur les nombres entiers naturels $\mathbb{N}$ est bien fondée
tandis que $<$ sur les nombres entiers relatifs $\mathbb{Z}$ ne l’est pas.

Dans ce cours, on se ramène aux nombres naturels par une **fonction de mesure** $m : \ldots \to \mathbb{N}$ (où $\ldots$ est le type avec lequel on travaille). On utilise souvent aussi un **variant** : une expression à resultat dans $\mathbb{N}$.

1.1.2 Récursion bien fondée

Une définition récursive est **bien fondée** ssi il existe une relation bien fondée entre les occurrences du terme défini telle que les occurrences qui apparaissent dans la définition sont plus simples que le terme défini.

Exemple :

```
factorielle(0) = 1
```

Il n'y a aucune occurrence dans la définition, donc elles sont toutes plus simples.

```
factorielle(n) = n * factorielle(n-1)  si n>0
```

L'occurrence `factorielle(n-1)` est plus simple que `factorielle(n)` si on utilise la valeur de l'argument comme mesure : `n-1 < n`.

1.1.3 Cas de base

Les éléments b qui n'ont pas d'élément "plus simple" sont appelés les "cas de base".

Par exemple :

– comme la mesure d'une liste est sa longueur, le cas de base d'un algorithme sur les listes est une liste de longueur 0, càd une liste vide.
– comme la mesure d'un arbre est sa hauteur, le cas de base d'un algorithme sur les arbres est une arbre de hauteur 0, càd un arbre vide.

1.1.4 Preuves par induction générale

Si pour tout élément,

en supposant qu'une propriété est vraie pour tous les élements plus simples,

on peut prouver qu'elle est vraie pour cet élément

alors

elle est vraie pour tous les éléments.

$$\frac{\forall x : \mathbb{N}\, (\forall y < x P(y)) \Longrightarrow P(x)}{\forall x : \mathbb{N}\, P(x)}$$

où $<$ est bien fondé dans $\mathbb{N}$.

Pour les cas de base b, il faut en fait prouver $P(b)$.

Pour les autres, dès qu'on tombe sur $P(y)$ plus simple, c'est prouvé !

1.1.5 Récursion croisée

Une récursion est **croisée** (ou **mutuelle**) si plusieurs termes définis forment un cycle.

Exemple (extrait du Larousse) :

Balayage : action de balayer.

Balayer : procéder à un balayage.

(Cet exemple est une récursion mal fondée.)

1.1.6 Fonctions récursives en Pascal

Les procédures et les fonctions récursives sont admises en Pascal. Les définitions par égalités (comme pour `factorielle`) ne sont pas admises, mais on les traduit par des **if then else**.

```
function factorielle(n:integer): integer;
{pré: n ≥ 0 }
{post: factorielle = n * (n−1) * ... * 2 * 1 }
{variant: n}
begin
  if n = 0
  then factorielle := 1
  else factorielle := n * factorielle(n−1)
end
```

1.1.7 Récursion croisée en Pascal

Les récursions croisées sont admises en Pascal. Il faut d'abord déclarer les procédures et fonctions comme **forward** pour que le compilateur connaisse leur type avant qu'elles ne soient utilisées :

```
function p1(n:integer): integer forward;
```

1.1.8 Procédure récursive : Exemple du labyrinthe

Exigences

Supposons qu'on reçoive un labyrinthe codé dans un tableau à deux dimensions dont les cases sont soit blanches (pour un couloir) ou 'X' pour un mur.

```
XXXXXXXXXXXXX
X  X  X        X
X  X     XXX  X
X  X  X     X  X
X     X   X  X
XXXXXXXXXXXXX
```

On demande d'imprimer un chemin vers la sortie.

Les objets utilisés par la procédure sont :

```
type xdim, ydim {intervalles};
     direction = nord,est,sud,ouest;
     labychar  = char { 'X', ' ', '.' };
     position  = record x: xdim; y: ydim end;
var  laby: array[xdim,ydim] of labychar;
```

La **recherche en profondeur** nous trouve un chemin de sortie récursivement : Elle marquera d'un '.' les cases blanches où on est déjà passé.

```
procedure sortir(pos: position);
var d: direction;
begin
  marquer(pos);
  if sortie(pos) then  sortietrouvée;
  else for d := nord to ouest
          if accessible(pas(pos,d)) then sortir(pas(pos,d));
end
```

Où `accessible` teste que la position n'est pas un mur et n'est pas déjà marquée ; `pas(pos,d)` avance d'un pas dans la direction `d`.

Base de la preuve : Le programme termine car le nombre de cases blanches (non marquées) diminue toujours. Il atteint toutes les cases accessibles car il essaie toutes les directions possibles.

Le chemin vers la sortie est constitué par les arguments de `sortir` ; on peut les conserver dans une liste pour les afficher.

1.2 Structures de données récursives

1.2.1 Liste

De même, on peut définir des structures de données récursivement.

Par exemple, une **liste** peut être définie comme étant soit une liste vide, soit composée d'un élément (la tête) et d'une liste (le reste).

Exemple : 1 — 3 — 7 — 5 — 3 est une liste ;

sa tête est 1 ; son reste est 3 — 7 — 5 — 3

1.2.2 Codage en Pascal

Les structures de données récursives sont interdites en Pascal, sauf pour les pointeurs. On les implémente donc par des pointeurs. Ce codage se dérive automatiquement de la définition récursive. Exemple :

```
type liste = ^ cell ;
     cell = record
              tete: information;
              reste: liste
                {inv: longueur(l^.reste) < longueur(l)};
            end;
```

Cette façon de coder les listes s'appelle les **listes (simplement) chaînées**.

Note : il faut ajouter l'invariant de données sur `liste`, car rien ne garantit en Pascal que les pointeurs sont sans cycles (bien fondés).

1.3 Arbres binaires

Un **arbre binaire** est soit un arbre vide, soit composé d'une information et de *deux* arbres (le fils gauche et le fils droit).

Une mesure bien fondée est la hauteur de l'arbre.

1.3.1 Arbres binaires en Pascal

```
type arbreBinaire = ^ cell;
     cell = record
              tete: information;
              gauche: arbreBinaire
                      {inv: hauteur(a^.gauche) < hauteur(a)};
              droit : arbreBinaire
                      {inv: hauteur(a^.droit)  < hauteur(a)};
            end;
```

1.3.2 Construire un arbre binaire

On peut construire n'importe quel arbre binaire au moyen de deux opérations :

```
function arbreVide: arbreBinaire;
begin
  arbreVide := nil;
end
```

```
function cons3(f: information; g,d: arbreBinaire): arbreBinaire;
var r: arbreBinaire;
begin
  new(r);
  r^.tete := f;
  r^.gauche := g;
  r^.droit := d;
  cons3 := r;
end
```

1.3.3 Observer un arbre binaire

```
function info(a: arbreBinaire): information;
begin
  info := a^.tete
end
function gauche(a: arbreBinaire): arbreBinaire;
begin
  gauche := a^.gauche
end
function droit(a: arbreBinaire): arbreBinaire;
begin
  droit := a^.droit
end
```

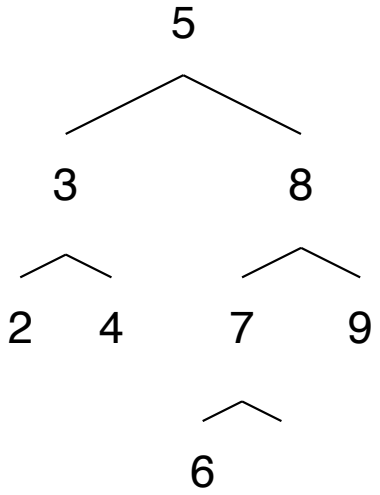
1.3.4 Recherche dans un arbre binaire

La fonction `dans` nous dit si une information se trouve dans l'arbre :

```
function dans(e:information; a: arbreBinaire): boolean;
dans(e, arbreVide) = false
dans(e, cons3(f,g,d)) = (e=f) or dans(e,g) or dans(e,d)
```

1.3.5 Arbres binaires triés (abt)

Un arbre binaire est **trié** ssi : il est vide ou son information est plus grande que toutes les informations contenues dans le fils de gauche et plus petite que toutes les informations contenues dans le fils de droite, et ses deux fils sont des arbres binaires triés.



1.3.6 Recherche dans un arbre binaire trié

La fonction `dans` nous dit si une information se trouve dans l'ABT :

```
function dans(e:information; a: abt): boolean;
```

$$\texttt{dans(e, arbreVide)} = \texttt{false}$$

$$\texttt{dans(e, cons3(f,g,d))} = \begin{cases} \texttt{true} & \texttt{si e = f} \\ \texttt{dans(e,g)} & \texttt{si e < f} \\ \texttt{dans(e,d)} & \texttt{si e > f} \end{cases}$$

On ne parcourt qu'une branche dans l'arbre.

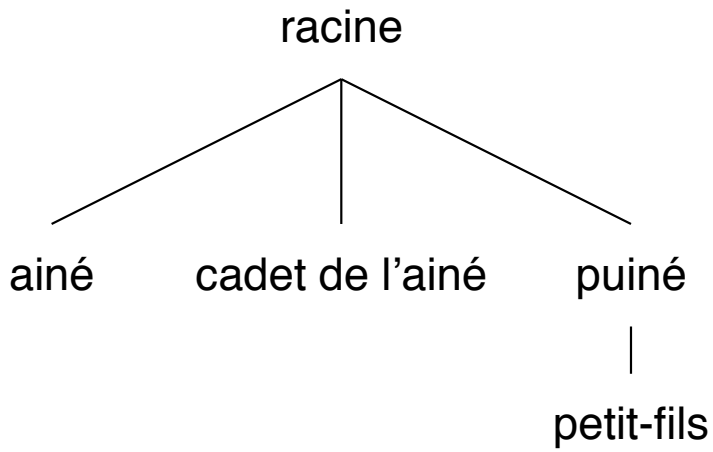
1.3.6.1 Recherche en Pascal

```
function dans(e:information; a: abt) : boolean;
begin
   if a = arbreVide
   then dans := false
   else if e = info(a)
        then dans := true
        else if e < info(a)
             then dans := dans(e,gauche(a))
             else dans := dans(e,droit(a))
end
```

1.4 Arbres ordonnés

un **arbre ordonné** est vide ou composé d'une information et d'une liste d'arbres (ses fils).

Le premier fils est appelé l'aîné ; le dernier, le puîné. Le suivant dans l'ordre est appelé le cadet.



1.4.1 Arbres ordonnés en Pascal

La définition récursive se code automatiquement en Pascal comme :

```
type arbreOrd = ^ cell;
     listearbreOrd = ^ celliste;
     cell = record
              e: information;
              fils: listearbreOrd;
            end;
     celliste = record
                  person: arbreOrd;
                  reste: listearbreOrd;
                end;
```

1.4.2 Représentation aîné/cadet

On identifie souvent le type des arbres et des listes d'arbres :

```
type listearbreOrd = ^ cell;
     arbreOrd = listearbreOrd; {inv(a): a <> nil}
      cell = record
                e: information;
                aîné: listearbreOrd;
                cadet: listearbreOrd;
             end
```

Notons que l'aîné représente (un pointeur vers) la liste des fils, tandis le cadet est l'habituel pointeur vers le reste de la liste.

Exercice : quel est l'invariant de représentation ?

CHAPITRE 2

TEMPS D'EXÉCUTION

2.1 Preuve de programmes

Pour calculer le temps d'un programme, on a besoin de son variant.

Voici un bref rappel de la preuve de programmes.

Notation	se lit	Auteur
$\{pre\}Prog\{post\}$	si on lance $Prog$ dans un état qui vérifie pre, il se termine dans un état qui vérifie $post$	Hoare
pre $\Longrightarrow$ wp(Prog,post)	plus faible précondition	Dijkstra
pre $\Longrightarrow$ [Prog] post	logique dynamique	Pratt, Harel
sp(Prog, pre) $\Longrightarrow$ post	plus forte postcondition	Dijkstra

La terminaison est incluse dans wp.

2.1.1 Règles de style

Pour que les règles de preuve simplifiées ci-après soient correctes, il faut que :

1. chaque fonction n'a pas **d'effet de bord** : elle ne modifie ni ses variables globales, ni ses paramètres par variable.
2. chaque fonction est **déterministe** : son résultat ne dépend que de la valeur de ses arguments.
3. il n'y a pas de variables **synonymes** (aliasing) : lors d'un appel de procédure, les paramètres par variable et les variables globales doivent désigner des emplacements distincts.
4. Chaque sous-programme p (fonction ou procédure) est décrite avec : sa pré- et post-condition, notées pre(p), post(p), son variant V_p si elle est récursive, les variables globales modifiées M_p et les éléments (variables, constantes, types) globaux lus L_p.

2.1.2 Expressions

Avant d'écrire une expression, il faut prouver sa pre !

La notation $[\vec{x} := \vec{e}]\phi$ signifie : la formule ϕ où on a remplacé chaque occurrence libre x_i d'une des variables dans $\vec{x}$ par l'expression correspondante e_i.

pre(e) est la précondition de e, définie par induction :

pre($f(\vec{e})$) = $[\vec{x} := \vec{e}]$pre(f) et pre($\vec{e}$)
pre($\vec{e}$) $= \bigwedge_{i=1}^{n}$ pre(e_i) = pre(e_1) et . . . et pre(e_n)
pre($a[i]$) = pre(i) et $bi \leq i \leq bs$ et init(a[i])
pre($a.c$) = pre(a) et init($a.c$)
pre(a↑) = pre(a) et not(free[a])
pre(x) = init(x)

init(x), où x est une variable, signifie « x est initialisé »càd « on a déjà mis une valeur dans x ». En Pascal, on ne peut lire une variable que si elle a été initialisée.

2.1.3 Affectation

$$\boxed{\text{wp}(x := E, P) = [x := E]P \text{ et pre}(E)}$$

càd en logique de Hoare

$$\{[x := E]P \text{ et pre}(E)\} \qquad x := E \qquad \{P\}$$

– x est un identificateur (nom) de variable.
– $[x := E]P$ signifie « P où l'on a remplacé les occurrences libres de x par E ».
– E est une expression du même type que x.

2.1.3.1 Affectation dans un tableau

$$\text{wp(a[i] := } e, R\text{) = pre}(i) \text{ et } bi \leq i \leq bs \text{ et pre}(e) \text{ et [a := a([i] } \rightarrow e)]R$$

où a est un nom de tableau ;
$a([i] \rightarrow e)$ signifie "le tableau a dont la case i a reçu la valeur e".

On peut généraliser l'indexation [i] à une suite s d'indexations et de sélection de champs,
p.ex. s = `[i].c[j]` si `a : array [1..N] of record c: array [1..N] of integer end`;

$$\text{wp(a } s := e, R\text{) = pre(a } s\text{) et pre}(e) \text{ et [a := a}(s \rightarrow e)]R$$

On peut simplifier par les règles :

$$\text{a([i] } s \rightarrow e\text{) [j] = if i=j then a[j]}(s \rightarrow e) \text{ else a[j]}$$

$$\text{Si } s \text{ est vide : a}(\rightarrow e) = e$$

$$\text{a(.c} \rightarrow e\text{).c} = e$$

$$\text{Si c} \neq \text{d : a(c} \rightarrow e\text{).d = a.d}$$

2.1.3.2 Pointeurs :

– La mémoire est un tableau `MT` indicé par les pointeurs de type `T` ;
– Un tableau de booléens `free` initialisé à `true` dit si une adresse n'est pas allouée. `free[nil]` restera toujours vrai.
– pre(`p^`) = `not free[p]`
– `p^` abrège `MT[p]`

new choisit un pointeur p libre et le réserve :

new(p : pointeur)

post : ($free_0$[p] = true) and (free = $free_0$([p] $\rightarrow$ false))

dispose(p)

pre : not free[p]

post : free = $free_0$([p] $\rightarrow$ true)

2.1.4 Séquence (;)

Précondition la plus faible :

wp(S_1 ; S_2, P) = wp(S_1, wp(S_2, P))

Hoare :

$$\frac{\{Q\}S_1\{R\} \quad \{R\}S_2\{P\}}{\{Q\}S_1; S_2\{P\}}$$

Logique dynamique :

$$[S_1; S_2]P = [S_1][S_2]P$$

2.1.5 Conditionnelle (if)

$$\frac{\{P \text{ et } B\}S_1\{Q\} \qquad \{P \text{ et non } B\}S_2\{Q\}}{\{P\} \textbf{ if } B \textbf{ then } S_1 \textbf{ else } S_2 \{Q\}}$$

$$\frac{\{P \text{ et } B\}S_1\{Q\} \qquad P \text{ et non } B \Rightarrow Q}{\{P\} \textbf{ if } B \textbf{ then } S_1 \{Q\}}$$

2.1.6 Boucle for

$$\frac{\{I \text{ et } l \leq i \leq h\}\ S\ \{[\text{i := succ}(i)]I\}}{\{[i := l]I\}\ \textbf{for}\ i := l\ \textbf{to}\ h\ \textbf{do}\ S\ \{[\text{i := max(succ}(h),l)]I\}}$$

on suppose que :

– la variable i et les bornes l, h ne sont pas modifiées par le corps de la boucle.

– les bornes l, h ne dépendent pas de i.

2.1.7 Boucle while

$$\frac{\{I \text{ et } B \text{ et } V = v_0\} \;\; S \;\; \{I \text{ et } V < v_0\} \qquad I \Rightarrow V \geq 0}{\{I\} \textbf{ while } B \textbf{ do } S \;\; \{I \text{ et non } B\}}$$

où

– V (le variant de boucle) est une expression qui borne le nombre d'itérations.
– v_0 est une variable logique, qui n'apparaît pas dans S, et retient la valeur précédente du variant.
– I (l'invariant de boucle) décrit ce qui reste vrai à chaque itération.
– $<$ est une relation bien fondée (ici, sur les nombres naturels).

Le programmeur doit donner V, I. Mais les méthodes de programmation (Chap. 6 et suivants) donneront cette information.

2.1.8 Appel de procédure

$\{I$ et ($[\vec{x} := \vec{a}]$ pre(p)) et pre$(\vec{a})$ et $V_p(\vec{a}) < v_0\}$ $p(\vec{a})$ $\{I$ et $[\vec{x} := \vec{a}]post(p)\}$

- $pre, post$ sont la pré- et post-condition de la procédure.
- $\vec{x}$ en sont les paramètres formels.
- $\vec{a}$ en sont les arguments effectifs.
- I (l'invariant de contexte) ne contient pas de variables modifiées par p.
- pour une procédure récursive :
  le variant de chaque appel récursif $V_p(\vec{a})$ doit être plus petit que celui v_0 du sous-programme q dans lequel il se trouve. v_0 est la valeur initiale de $V_q(\vec{x})$.

2.1.9 Appel de fonction

$$[x := \vec{a}]pre \Longrightarrow [f := f(\vec{a}), x := \vec{a}]post$$

où

– $pre, post$ sont la pré- et post-condition de la fonction.
– $\vec{x}$ en sont les paramètres formels.
– $\vec{a}$ sont les arguments effectifs.
– le variant d'un appel récursif $V_f(\vec{a})$ doit être plus petit que v_0, la valeur initiale du variant du sous-programme q dans lequel il se trouve.
– Cet axiome peut être utilisé n'importe où dans une preuve.

2.2 Ordre de grandeur

1. Le temps d'exécution dépend de la taille des données :
   Soit n cette taille,
   on veut une expression $T_P(n)$ du temps d'exécution du programme P.
2. Le temps d'exécution varie d'un facteur plus ou moins constant d'un ordinateur à un autre.
3. La temps d'exécution pour les petites données est négligeable
   $\Rightarrow$ On calcule seulement l'ordre de grandeur $\mathcal{O}$ de $T(n)$, à une constante près, et pour les données assez grandes (asymptotique).

$$\mathcal{O}(T(n)) \leq \mathcal{O}(f(n)) \Longleftrightarrow \exists c, n_0 > 0 : \forall n > n_0 : T(n) \leq c * f(n)$$

2.2.1 Calcul avec $\mathcal{O}$.

$$\mathcal{O}(f(n)) = \mathcal{O}(g(n))$$

Signifie :

$$\mathcal{O}(f(n)) \leq \mathcal{O}(g(n)) \wedge \mathcal{O}(g(n)) \leq \mathcal{O}(f(n))$$

Note : La notation officielle est $f(n) = \Theta(g(n))$.

2.2.1.1 Théorèmes

Si la variable $n \geq 0$, les constantes $k, l \geq 0$ et $f(n) \geq 0$:

$$\mathcal{O}(k.f(n)) = \mathcal{O}(f(n))$$

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n)) \quad \text{si } \mathcal{O}(g(n)) \leq \mathcal{O}(f(n))$$

$$\mathcal{O}(f(n)^k) \leq \mathcal{O}(f(n)^l) \quad \text{si } 0 \leq k \leq l$$

$$\mathcal{O}(n^l) \leq \mathcal{O}(k^n) \quad \text{si } k > 1$$

$$\mathcal{O}(l^n) \leq \mathcal{O}(k^n) \quad \text{si } k > l$$

$$\mathcal{O}(f(n)) * \mathcal{O}(g(n)) = \mathcal{O}(f(n) * g(n))$$

$$\mathcal{O}(max(f(n), g(n))) = \mathcal{O}(f(n)) \quad \text{si } \mathcal{O}(g(n)) \leq \mathcal{O}(f(n))$$

2.3 Règles pour le temps d'exécution

Pour chaque forme de programme, on a une règle de calcul de son temps d'exécution. On peut donc calculer mécaniquement le temps de tout programme non récursif.

2.3.1 Séquence

$$\boxed{T_{S_1;S_2} = T_{S_1} + T_{S_2}}$$

Le temps pour faire la séquence d'opérations S_1 puis S_2 est la somme des temps d'exécution de S_1 et de S_2.

2.3.2 Affectation

$$\boxed{T_{x:=E} = T_E + T_A}$$

Le temps pour calculer une affectation simple est le temps d'évaluer l'expression T_E plus le temps T_A de mettre le résultat dans la variable.

Dans le modèle RAM (random access memory), lire ou écrire un type simple (entier, réel, booléen, caractère, énuméré) ou pointeur prend un temps constant.

Les types composés (tableaux, chaînes de caractères, etc.) prennent un temps proportionnel à leur taille mémoire.

2.3.3 Appel

$$\boxed{T_{f(e_1,\ldots,e_n)} = \sum_i T_{e_i} + \mathcal{O}(1) + T_f(V_f(e_1,\ldots,e_n))}$$

Le temps d'évaluer un appel de sous-programme est le temps d'évaluer ses arguments, d'appeler le sous-programme, d'exécuter le corps du sous-programme.

V_f est le variant de f : une fonction des paramètres de f qui en mesure la complexité.

On suppose $T_f = \mathcal{O}(1)$ pour les opérations prédéfinies sur les types simples (addition, etc.)

2.3.4 Conditionnelle (if)

$$\boxed{T_{\text{if } B \text{ then } S_1 \text{ else } S_2} = T_B + \text{if } B \text{ then } T_{S_1} \text{ else } T_{S_2}}$$

Pour faire le test, il faut :

1. évaluer B
2. suivant sa valeur, faire S_1 ou S_2.

En pratique il est plus facile d'employer la borne supérieure :

$$\boxed{T_{\text{if } B \text{ then } S_1 \text{ else } S_2} \leq T_B + max(T_{S_1}, T_{S_2})}$$

Exemple :

$T_{\text{if false then while } \ldots}$ = $\mathcal{O}(1)$ par la règle exacte

$\leq \mathcal{O}(T_{while\ldots})$ par la règle approchée.

2.3.5 Boucle for

$$\boxed{T_{\text{for } i \text{ :=} l \text{ to } h \text{ do } S} = T_l + T_h + \text{ if } l \leq h \text{ then } (h - l + 1) * (T_S + \mathcal{O}(1)) \text{ else } 0}$$

On calcule l et h au début, puis il faut faire $(h - l + 1)$ fois le corps ainsi que le test et l'incrément ($\mathcal{O}(1)$).

NB : On a souvent :

1. $T_S \geq \mathcal{O}(1)$ et donc $\mathcal{O}(T_S + \mathcal{O}(1)) = \mathcal{O}(T_S)$.
2. T_l et $T_h \leq \mathcal{O}(h - l + 1) * T_S$, on peut alors négliger ces termes.
3. $l \leq h$

On emploie alors $T_{\text{for } i \text{ :=} l \text{ to } h \text{ do } S} = \mathcal{O}((h - l + 1) * T_S)$

2.3.6 Boucle while

Comment savoir combien de fois la boucle est exécutée ?

on trouve un **variant** V qui diminue à chaque passage dans la boucle. Sa valeur initiale est une borne supérieure du nombre d'itérations.

$$\boxed{T_{\text{while } B \text{ do } S} \leq (V_0 + 1) * T_B + V_0 * T_S}$$

La formule devient une égalité si le variant diminue de 1 à chaque passage, et qu'on sort de la boucle ssi le variant vaut 0.

Le plus souvent, $T_B \leq \mathcal{O}(T_S)$, donc on emploie $\leq \mathcal{O}(V_0 * T_S)$, dite "nombre de passages fois le temps d'un passage".

2.3.7 Temps des sous-programmes récursifs

La règle générale s'applique mais donne une équation récurrente.

Attention $\mathcal{O}$ ne s'applique qu'à une fonction déjà définie.
Pour les appels récursifs, il ne faut pas employer $\mathcal{O}$.

Ne pas supprimer les constantes

$T_1(n) = T_1(n-1) + k, T_1(0) = k$ a comme solution $T_1(n) = k.(n+1)$ (temps linéaire), tandis que $T_2(n) = 2.T_2(n-1)$ a comme solution $T_2(n) = k.2^n$ (temps exponentiel), même si $\mathcal{O}(f(n)+k) = \mathcal{O}(2f(n))$ pourrait faire croire que ces deux équations donneraient le même ordre de grandeur.

$T_1(n) = T_1(n-1) + k$	$T_2(n) = 2.T_2(n-1)$
k	k
$T_1(1) = 2.k$	$T_2(1) = 2.k$
$T_1(2) = 3.k$	$T_2(2) = 4.k$
$T_1(3) = 4.k$	$T_2(3) = 8.k$
...	...
$T_1(n) = (n+1).k$	$T_2(n) = 2^n.k$

Ne pas induire sur l'ordre de grandeur

$$T(n) = T(n-1) + \mathcal{O}(1)$$
$$T(0) = \mathcal{O}(1)$$

On pourrait croire que

$$T(0) = \mathcal{O}(1)$$
$$T(1) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$
$$T(2) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$
$$\dots$$

et que par "induction" :

$T(n) = \mathcal{O}(1)$, ce qui est évidemment faux puisque $T(n) = \mathcal{O}(n)$.

2.3.7.1 Équations récurrentes

équation récurrente	où...	$T(n) = \mathcal{O}(\ldots)$
$T(n) = cT(n-1) + bn^k$	$c = 1$	n^{k+1}
"	$c > 1$	c^n
$T(n) = cT(n/d) + bn^k$	$c > d^k$	$n^{\log_d c}$
"	$c = d^k$	$n^k \log n$
"	$c < d^k$	n^k

2.3.8 Espace en mémoire

L'espace dépend du type de chaque variable :

- type simple ou pointeur : espace constant ;
- `array` : espace proportionnel au produit des tailles de ses indices, fois l'espace du type élément, où la taille d'un type indice `l..u` est `max(u-l+1,0)`
- `record` : somme des tailles des champs ;
- `set` : habituellement, on utilise les tableaux de booléens et l'espace est donc le nombre de valeurs possibles du type de base (voir chapitre 4).

Pour les sous-programmes récursifs, on crée une copie des paramètres et variables locales à chaque appel récursif : il faut donc multiplier l'espace de ceux-ci par la profondeur d'appel, qui est bornée par la valeur initiale du variant.

Pour les cellules du tas, on calcule le nombre de cellules actives, càd le nombre d'appels à `new` mois le nombre d'appels à `dispose` pour ce type, et on le multiplie par l'espace pris par une cellule.

2.3.9 Exemples

2.3.9.1 Primalité

```
function premier(n: integer): boolean;
var i : integer;
    p : boolean; {= premier}
begin
     p := true;  i := 2;
     while sqr(i) <= n and p do
     begin
          p :=  (n mod i <> 0)
          i := i + 1;
     end;
     premier := p;
```

end ;

Pré : n > 0

Post : premier = $\nexists$ i : i divise n et 1 < i < n

Invariant : 1<i<$\sqrt{n}$ et p = $\forall$ j . 1<j<i : j ne divise pas n

Variant : $\lfloor \sqrt{n} \rfloor - i$

Temps : $T_{premier}(n) = \mathcal{O}(\sqrt{n})$

Espace : $E_{premier}(n) = \mathcal{O}(1)$

2.3.9.2 Fibonacci

Définition

$$Fib(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

$n \in \mathbb{N}$

Définition récursive

$$Fib(0) = 0$$
$$Fib(1) = 1$$
$$Fib(2) = 1$$
$$Fib(3) = 2$$
$$Fib(4) = 3$$
$$Fib(5) = 5$$
$$Fib(6) = 8$$
$$\ldots$$

En général, pour $n > 1$:

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Preuve On pose :

$$
\begin{aligned}
a &= \frac{1}{\sqrt{5}} \\
O_1 &= \frac{1+\sqrt{5}}{2} \\
O_2 &= \frac{1-\sqrt{5}}{2} \\
g_1 &= a.O_1^n \\
g_2 &= a.O_2^n \\
d_1 &= a(O_1^{n-1} + O_1^{n-2}) \\
d_2 &= a(O_2^{n-1} + O_2^{n-2})
\end{aligned}
$$

Notons que O_1, O_2 sont les racines de $x^2 = x + 1$ (les nombres d'or).

$$Fib(n) = aO_1^n - aO_2^n \qquad = g_1 - g_2$$
$$Fib(n-1) + Fib(n-2) = aO_1^{n-1} - aO_2^{n-1}) + aO_1^{n-2} - aO_2^{n-2} \qquad = d_1 - d_2$$

il suffit de prouver $g_1 = d_1$ et $g_2 = d_2$, càd

$$aO_1^n = aO_1^{n-1} + aO_1^{n-2})$$

mettons aO_1^{n-2} en évidence, il reste :

$$O_1^2 = O_1 + 1$$

```
function Fib(n: integer): integer;
{Pré: n ≥ 0;
 Variant: n }
begin
     if n < 2 then Fib := n
     else Fib := Fib(n−1) + Fib(n−2)
end;
```

Équation récurrente

$$T(0) = T(1) = c$$
$$T(n) = T(n-1) + T(n-2) + b, \text{ pour } n > 1$$

Solution :

$$\begin{aligned} T(n) &= c * Fib(n+1) + b * (Fib(n+1) - 1) \\ &= \mathcal{O}(O_1^n) \text{ temps exponentiel} \end{aligned}$$

Espace : 1 argument en espace constant * variant n = $\mathcal{O}(n)$ espace linéaire

2.3.9.3 Position du Maximum

Calculons le temps d'exécution de :

```
function posMax(var a: tableau; bi, bs: integer) : integer;
{Pré: bi ≤ bs
      a[bi..bs] initialisé
Post: pour tout i dans bi..bs, a[posMax] >= a[i]
       posMax dans bi..bs  }
begin
      if bi >= bs then posMax := bi
      else if a[bi] < a[posMax(a,bi+1,bs)]
           then posMax := posMax(a,bi+1,bs)
           else posMax := bi
end;
```

Soit n la taille du tableau : $n = bs - bi + 1$. Pour $n = 1$, $bs = bi$, on exécute le then du premier test. Pour $n > 1$, $bs > bi$, on exécute le else du premier test.

$$\begin{aligned} T(1) &= \mathcal{O}(1) \\ T(n) &= \mathcal{O}(1) + T(n-1) + \max(T(n-1) + \mathcal{O}(1), \mathcal{O}(1)) \\ &= \mathcal{O}(1) + 2T(n-1) \text{ si } n > 1 \end{aligned}$$

Et donc :

$$\Rightarrow T(n) = \mathcal{O}(2^n)$$

Très inefficace !

```
function posMax(var a: tableau; bi, bs: integer) : integer;
var posReste : integer;
begin
     if bi ≥ bs then posMax := bi
     else begin
          posReste := posMax(a,bi+1,bs);
          if a[bi] < a[posReste]
             then posMax := posReste
             else posMax := bi
          end
end;
```

$$
\begin{aligned}
T(1) &= \mathcal{O}(1) \\
T(n) &= \mathcal{O}(1) + T(n-1) + \max(\mathcal{O}(1), \mathcal{O}(1)) \; (n > 1) \\
&= T(n-1) + \mathcal{O}(1)
\end{aligned}
$$

$$
\Rightarrow T(n) = \mathcal{O}(n)
$$

2.3.9.4 Tri par sélection

```
procedure tri(var a: tableau);
var i: integer;
begin
      for i := n downto 1 do
          echange(a[i],a[posMax(a,1,i)]);
end;
```

$$\begin{aligned} T(n) &= b.(n + (n-1) + \ldots + 2 + 1) \\ &= b \sum_{i=1}^{n} i \\ &= b \frac{n(n+1)}{2} \\ &= \mathcal{O}(n^2) \end{aligned}$$

2.3.9.5 Tri par fusion

Supposons que `fusion(a,bi,m,bs)` s'exécute en temps $\mathcal{O}(bs - bi) = \mathcal{O}(n)$.

```
procedure tri(var a: tableau; bi,bs: integer);
var c : integer; {centre du tableau}
begin
      if bs > bi                    {O(1)}
      then begin
           c := (bi+bs) div 2;      {O(1)}
           tri(a,bi,c);             {T(n/2)}
           tri(a,c+1,bs);           {T(n/2)}
           fusion(a,bi,c,bs);        {O(n)}
      end;
end;
```

Qui donne l'équation récurrente :

$$T(n) = 2T(n/2) + bn$$

$$\Rightarrow T(n) = \mathcal{O}(n \log n) < \mathcal{O}(n^2)$$

Cet algorithme est beaucoup plus rapide !

CHAPITRE 3

INTRODUCTION AUX TYPES ABSTRAITS

3.1 Définition

Type Abstrait =

- un type
- des sous-programmes (procédures et fonctions) qui utilisent ce type
- des propriétés de ces sous-programmes.

$\neq$

Type concret = un type défini par une combinaison de types de base.

3.1.1 Exemple : Pascal

Tous les types de base de Pascal peuvent être considérés commes de types abstraits pourvu qu'on donne leurs propriétés.

```
type integer

function + (x,y : integer) : integer
function − (x,y : integer) : integer
function * (x,y : integer) : integer
function div (x,y : integer) : integer

const 0 : integer
      ...
```

Bien sûr, ceci n'est pas syntaxiquement correct en Pascal !

3.1.2 Exemple : Les listes

Une liste contient une suite finie d'éléments : le premier, le deuxième, ... On se donne des fonctions :

- `listeVide` construit une liste vide
- `cons(x,l)` construit une nouvelle liste en ajoutant `x` devant `l`
- `head(l)` donne le premier élément de la liste
- `nth(i,l)` donne le i-ème élément de la liste
- `tail(l)` donne le reste de la liste (le deuxième, ...)
- `append (l1,l2)` donne une liste formée de `l1` suivi de `l2`, càd la *concaténation* de `l1` et `l2`.
- `length(l)` la longueur de la liste

3.2 Avantages

1. Abstraction : On peut construire un programme utilisant le type abstrait sans connaître son implémentation.
2. Modifiabilité : On peut modifier l'implémentation du TA, pourvu que les propriétés restent vraies, SANS devoir changer le programme utilisateur.
3. Encapsulation : Les données ne sont changées que par les sous-programmes de l'interface, ce qui garantit un invariant des données
4. Réutilisation : Les types abstraits sont employés dans un grand nombre de programmes.

3.3 Syntaxe

Il n'y a pas de syntaxe pour les Types Abstrait en Pascal standard, mais bien dans ses extensions.

Language		Abstrait Visible	Concret Caché
Turbo Pascal	unité	interface	implémentation
Ada	paquetage	spécification	corps
Modula-2	module	définition	implémentation

En Turbo Pascal

Une unité (`unit`) consiste en

– une interface, qui déclare les constantes, les types, et les en-têtes de fonctions et de procédures ;
– suivie d'une implémentation, qui contient le corps des fonctions et procédures.

Problème : une interface ne correspond pas exactement à un type abstrait : on y trouve le définition concrète du type, qui est liée à l'implémentation.

Free Pascal et Delphi sont basés sur la même syntaxe.

Exemple en Turbo Pascal

```
Unit Ensemble
Interface
    const ensVide = nil; (* ! implémentation *)
    type ensemble = ^cell; (* ! implémentation *)
    function union(x,y : ensemble) : ensemble;
Implementation
    uses Listes;
    function union(x,y :ensemble) : ensemble;
    begin
          ...
    end;
end.
```

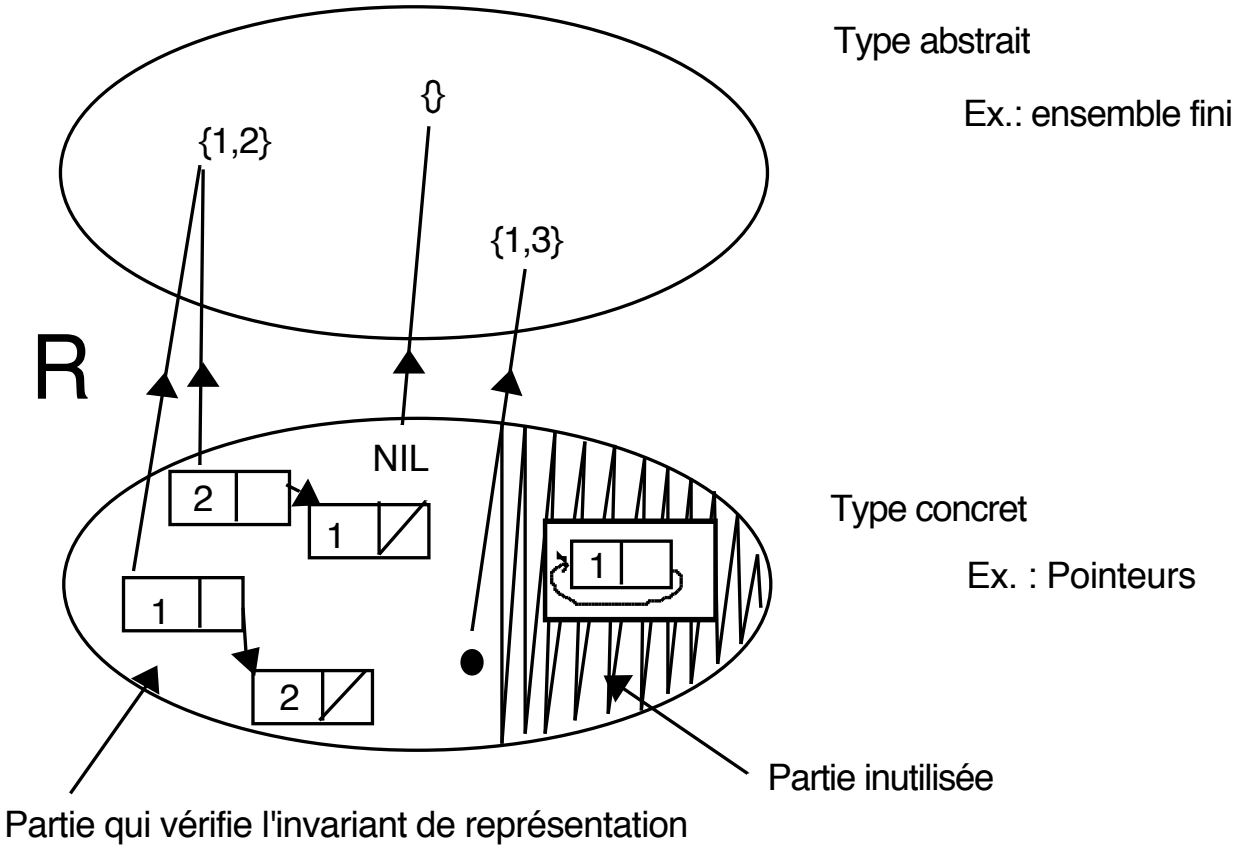
3.4 Comment donner les propriétés

3.4.1 Par modèles [VDM, Z, B]

1. On donne une “implémentation” abstraite basée sur les ensembles
2. Les sous programmes sont spécifiés par Pré/Post-conditions (sur l’implémentation abstraite).

3.4.1.1 Raffinement

Pour prouver une implémentation concrète C, on définit un **invariant de représentation** (ou **d'abstraction** ou **de collage**) $R(a, c)$ entre le type abstrait A et le type concret C.



L'**invariant de données** décrit la partie des données concrètes utilisées

$$D(c) = \exists a, R(a, c)$$

Si ce a est unique, on définit une fonction *d'abstraction* Info : $C \to A$
et $R(a, c)$ peut s'écrire comme : $D(c)$ et $a = \text{Info}(c)$

Pour chaque sous-programme concret $f_C(c : C; b : B) : C$, on suppose l'invariant de données dans la précondition, mais on doit le prouver (rétablir) dans la post-condition. Les pré et post-conditions abstraites doivent être traduites avec Info. La pré-condition concrète est donc :

$$D(c) \text{ et } \text{pre}(f_A(\text{Info}(c), b))$$

Et la post-condition concrète :

$$D(f_C(c, b)) \text{ et } \text{post} f_A(\text{Info}(c), b))$$

3.4.1.2 Exemple : TA listes par modèle

Les listes sont définies comme un ensemble de couples (indice dans la séquence, valeur) avec les propriétés :

1. il n'y a pas de valeur à l'indice 0 (on commence à 1)
2. il ne peut y avoir deux valeurs au même indice
3. les indices vont de 1 à n

P.ex. la liste $(5, 4, 3)$ est représentée par l'ensemble $\{(1, 5), (2, 4), (3, 3)\}$

Type liste = $\mathcal{P}(N \times elem)$

tel que $\nexists e : (0, e) \in l$

$\nexists i, e_1, e_2 : (i, e_1) \in l \; et \; (i, e_2) \in l \; et \; e_1 \neq e_2$

$\forall (i, e) \in l, \; i > 1 \Rightarrow \exists e_2 : (i - 1, e_2) \in l$

function listeVide : liste

Post : listeVide = {}

function cons(x : elem ; l : liste) :liste

Post : cons$(x, l) = \{(1, x)\} \bigcup \{(i + 1, y) | (i, y) \in l\}$

function head(l : liste) :elem

Pré : $l \neq \{\}$ (ou $\exists x : (1, x) \in l$)

Post : head = $x \Leftarrow (1, x) \in l$

function tail(l : liste) : liste

Pré : $l \neq \{\}$

Post : tail = $\{(i, x) | (i + 1, x) \in l \; et \; i \geq 1\}$

function null(l : liste) : boolean

Post : null = $(l = \{\})$

function append(l_1, l_2 : liste) : liste

Post : append = $l_1 \bigcup \{(n + i, x) | (i, x) \in l_2, n = |l_1|\}$

3.4.2 Par axiomes

On donne des propriétés en termes des sous-programmes déclarés dans l'interface.

Pour les fonctions (sans effet de bord) On emploie la logique du premier ordre.

Exemple : $\forall x : integer; A, B : ensemble$:

$$dans(x, union(A, B)) \Longleftrightarrow (dans(x, A)\ ou\ dans(x, B))$$

Pour les procédures On ajoute la logique dynamique :

$[Prog]Formule$ signifie : Après toute exécution de Prog, la formule est vraie.

Exemple : $[ajouter(x, A)]dans(x, A)$

Complétude

Comment être sûr d'avoir donné assez d'axiomes ? Lorsqu'il n'y a que des fonctions : 2 méthodes :

3.4.2.1 Méthode des constructeurs

1. Choisir un sous-ensemble des fonctions qui permet de construire toutes les valeurs du TA (les constructeurs).
2. Donner toutes les combinaisons de la forme

$$n(c_1(\ldots), c_2(\ldots)) = e$$

où

   - n est une fonction, mais pas un constructeur
   - c_1, c_2 sont des constructeurs.
   - e est une expression plus simple (pour un ordre bien fondé).

3. Puis les “équations entre constructeurs” de la forme

$$c_1(c_2(\ldots)) = e$$

Exemple : Listes : constructeurs

Constructeurs : { listeVide, cons}

Exemple : (1,2) = cons(1,cons(2,listeVide)).

Axiomes par la méthode des constructeurs.

- head(listeVide) = indéfini.
- head(cons(x,l)) = x.
- tail(listeVide) = indéfini.
- tail(cons(x,l)) = l.
- null(listeVide) = true.
- null(cons(x,l)) = false.
- append(listeVide,l2) = l2.
- append(cons(x,l),l2) = cons(x,append(l,l2)).

3.4.2.2 Méthode des observateurs

– Choisir un sous-ensemble de fonctions qui permet d'observer toute différence entre 2 valeurs.
– Donner toutes les combinaisons de la forme

$$o(n(x)) = e \ldots o(x) \ldots$$

où

– o est un observateur.
– n est une fonction non observateur.
– e est une expression qui ne contient pas $o(x)$.

Observateurs : { head, tail, null}.

head(listeVide) = indéfini.

tail(listeVide) = indéfini.

null(listeVide) = true.

head(cons(x,l))= x.

tail(cons(x,l)) = l.

null(cons(x,l)) = false.

$$\text{head(append(l1,l2))} = \begin{cases} \text{head(l1)} & \text{si null(l1) = false.} \\ \text{head(l2)} & \text{si null(l1) = true.} \end{cases}$$

$$\text{tail(append(l1,l2))} \begin{cases} = \text{append(tail(l1),l2)} & \text{si null(l1) = false.} \\ = \text{tail(l2)} & \text{si null(l1) = true.} \end{cases}$$

null(append(l1,l2)) = null(l1) and null(l2).

3.4.3 Implémentation par axiomes

Soit A un type abstrait, C le type concret qui l'implémente, D la sous-algèbre de C qu'on emploie. On doit montrer que la fonction d'abstraction `Info` est un homomorphisme de $D \rightarrow A$, càd :

1. les fonctions concrètes restent dans D (préservent l'invariant de données) :
   $\forall c \in C : D(c) \Longrightarrow D((f_C(c)))$
2. Les axiomes sont vérifiés par les fonctions concrètes : si
   $\forall x_1, x_2, \ldots x_n : A, r = l$ est un axiome, alors on doit prouver :
   $\forall x_1, x_2, \ldots, x_n : D$
   $Info([f_A := f_C]r = Info([f_A := f_C]l)$
   où $[f_A := f_C]r(c)$ signifie $r(a)$ où toutes les fonctions abstraites ont été remplacées par leur correspondant concret.

3.5 Le type abstrait “Pile”

Une pile est une collection d’éléments récupérée dans l’ordre inverse de celui où on les a mises : en anglais “Last In, First Out” (LIFO)

Fonctions :

1. `pileVide` donne une pile vide
2. `empile` (ou `push`) ajoute un élément en sommet de pile
3. `dépile` (ou `pop`) retire le sommet de pile
4. `sommet` (ou `top`) renvoie le sommet de pile
5. `hauteur` (ou `top`) renvoie le nombre d’éléments dans la pile

3.5.1 Axiomes

– sommet(pileVide) = indéfini.
– sommet(empile(x,l)) = x.
– dépile(pileVide) = indéfini.
– dépile(empile(x,l)) = l.
– hauteur(pileVide) = 0
– hauteur(empile(x,l)) = 1 + hauteur(l)

Une pile *bornée* a de plus l’axiome : empile(x,l) = indéfini si hauteur(l) = max

3.5.2 Implémentation : Tableau+pointeur

Implémentation classique des piles.

```
type pile = record
                a: array[1..max] of elem;
                p: 0..max
            end
```

– Cette implémentation convient mieux en procédural

– En Pascal, on doit borner le tableau : La pile doit donc aussi être bornée.

– La fonction d’abstraction A est récursive :

A(x) = pileVide si x.p = 0

A(x) = empile(x.a[x.p], A(x(p $\rightarrow$ p-1)) si x.p > 0

3.6 TA File de priorité

3.6.1 Spécification par modèle

procedure inserer(E: ens; x: elem)

Post: $E = E_0 \cup \{x\}$

procedure SupprimerMax(E: ens; **var** m: elem)

Pré: E non vide.

Post: m = max(E0)

E = E0 \ {m}

3.7 Implémentation

3.7.1 Arbre binaire partiellement ordonné (APO)

Invariant Un père est plus grand que ses fils.

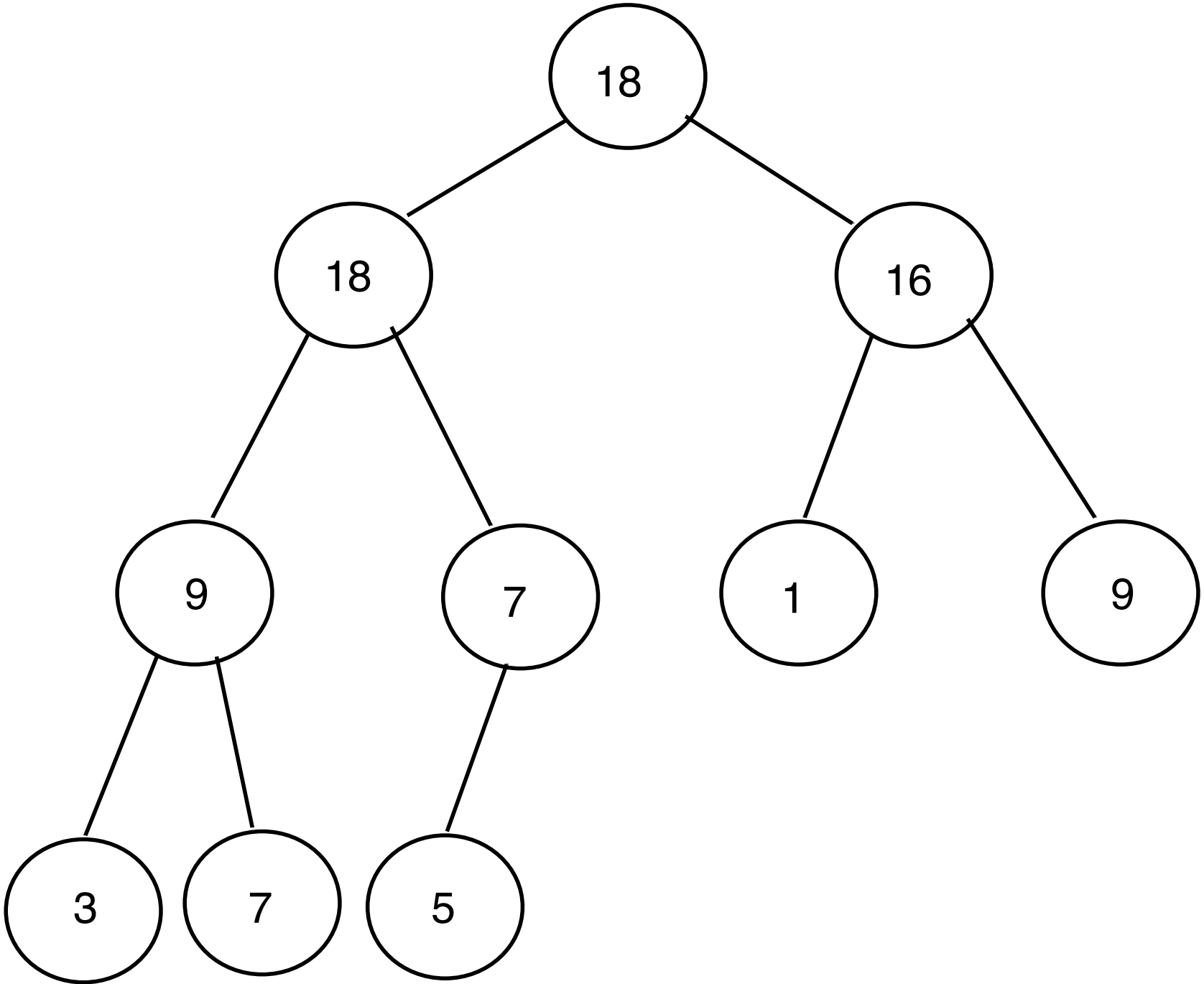
3.7.2 Tas

Invariant

De plus :

1. tous les niveaux, sauf le dernier, sont remplis.
   càd : tous les chemins ont la même longeur à 1 près.
2. le dernier niveau est rempli d'abord à gauche.

Exemple un arbre partiellement ordonné avec 10 noeuds.


18
18
16
9
7
1
9
3
7
5

Qui est représenté par le tableau `A` + `n` le nombre d'éléments.

```
type tas = record
                A: array[1..MAX] of elem;
                n: natural; {nombre d'éléments dans le tas }
           end;
```

i :	1	2	3	4	5	6	7	8	9	10
A[i] :	18	18	16	9	7	1	9	3	7	5

Invariant de données : $0 \leq n \leq$ MAX et $\forall i : 1 \leq i$ et
$2 * i \leq n \Rightarrow A[i] \geq A[2 * i]$ et $\forall i : 1 \leq i$ et
$2 * i + 1 \leq n \Rightarrow A[i] \geq A[2 * i + 1]$

3.7.3 Procédures et fonctions

L'insertion dans un tas.

```
procedure swap(var x,y  : elem);
var temp : elem;
begin
     temp := x;
     x := y;
    y := temp;
end;
```

```
procedure bubbleUp(var t: tas; i: integer);
begin
     with t begin
     if i > 1
     then if A[i] > A[i div 2]
          then begin
               swap(A[i], A[i div 2]);
               bubbleUp(A, i div 2)
          end
     end
end;
```

```
procedure inserer( x : elem; var t : tas);
begin
   with t begin
     n := n + 1;
     A[n] := x;
     bubbleUp(t, n);
  end
end;
```

$bubbleDown$ fait descendre un élément i responsable de la violation de la propriété APO jusqu'à ce qu'une feuille soit atteinte.

```
procedure bubbleDown(var t: tas; i : integer);
var child : integer;
begin
   with t begin
     child := 2 * i;
     if child < n
     then if A[child + 1] > A[child]
          then child := child + 1;
     if child <= n
     then if A[i] < A[child]
          then begin
               swap(A[i], A[child]);
```

```
                bubbleDown(t, child);
            end
        end
    end;
```

```
procedure deletemax(var t: tas; var x:elem);
begin
   with t do begin
     x := A[1]
     swap(A[1], A[n]);
     n := n − 1;
     bubbleDown(t, 1);
end;
```

Transformer un tableau en tas. Le type `tableau_n` a la même déclaration qu'un tas, moins l'invariant APO.

```
procedure heapify(var t : tableau_n);
var i : integer;
begin
     for  i := n div 2 downto 1 do
          bubbleDown(t , i);
end;
```

Tri par tas dans un tableau.

```
procedure heapsort(var t : tableau_n);
begin
     heapify(t);
     while n > 1 do  deletemax(t,A[n])
end;
```

3.8 Types Abstrait Ensemble fini

NB : ce type fait partie du Pascal, mais il est souvent inefficace.

```
type ens     {set of elem}                    ; { Notation en Pascal }

function singleton (e:elem): ens              ; { [e] }
function ensVide : ens                        ; { [] }
function union (x,y: ens): ens                ; { [x+y] }
function intersection (x,y: ens): ens         ; { [x*y] }
function ajout (e: elem; x :ens): ens         ; { [[e]+ x] }
function dans (e: elem; x :ens): boolean      ; {e in x}
```

3.8.1 Axiomes

Constructeurs : {ensVide, ajout}.

- singleton(e) = ajout(e, ensVide).
- union(ensVide,y) = y.
- union(ajout(e,x),y) = ajout(e,union(x,y)).
- intersection(ensVide,y) = ensVide.
- intersection(ajout(e,x),y) =
  - intersection(x,y) si dans(e,y) = false.
  - ajout(e,intersection(x,y)) si dans(e,y)= true
- dans(e,ensVide) = false.
- dans(e, ajout(e,x))= true.
- dans(e,ajout(f,x)) = (e=f) ou dans(e,x).

{Equations entre constructeurs}

- ajout(e,ajout(e,x)) = ajout(e,x).
- ajout(e,ajout(f,x)) = ajout(f,ajout(e,x)).

3.9 Type Abstrait Dictionnaire

Ce TA permet de retrouver la “définition” d’un “mot”, aussi appelé “clé”.

On l’appelle aussi `Map` en Java 2.

```
type dico

function ajout(d : def; m : mot; di : dico) : dico;
         dicoVide : dico;
         def_de(m : mot; di : dico) : def;
         dans(m : mot; di : dico) : boolean;
         après(di1,di2:dico):dico;
```

Toutes nos implémentation d’ensembles peuvent être adaptées pour le type dictionnaire.

3.9.0.1 Axiomes par Constructeurs

- dans(m, dicoVide) = false.
- dans(m, ajout(d, n, di)) = m = n ou dans(m, di)
- def_de(m, dicoVide) = indéfini.
- def_de(m, ajout(d, m, di)) = d.
- def_de(m, ajout(d, n, di)) = def_de(m, di) (pour n $\neq$ m).
- après(dicoVide, di2) = di2.
- après(ajout(d, m, di), di2) = ajout(d, m, après(di, di2)).
- ajout(d2, m, ajout(d1, m, di)) = ajout(d2, m, di).
- ajout(d2, m, ajout(d1, n, di)) = ajout(d1, n, ajout(d2, m ,di)) (pour n $\neq$ m).

3.9.1 Ensembles avec procédures

```
type ens = P(elem);

procedure insérer(e: elem; var x: ens);
        Post : x = x0 + [e]

procedure supprimer(e: elem; var x: ens);
        Post : x = x0 - [e]
```

3.9.2 Dictionnaire avec procédures

type ens = *{\cal P}*(mot $\times$ def);

procedure insérer(m: mot; d: def; **var** x: dico);

Post : x = après($\{m \mapsto d\}, x_0$)

procedure supprimer(m: mot; **var** x: dico);

Post : $x = x_0 \setminus \{m \mapsto d | d \in def\}$

CHAPITRE 4

IMPLÉMENTATION DES ENSEMBLES

Différentes structures vont être abordées :

1. Tableau de booléens.
2. Liste avec doubles.
3. Liste sans doubles.
4. Liste triée.
5. Table de hachage.
6. Arbre binaire de recherche.
7. Arbre rouge/noir.
8. B-Arbre.

4.1 Tableau de Booléens

Cette implémentation, aussi appelée “vecteur de bits” est définie comme :

type `ens` = **packed array**[`elem`] **of boolean**

invariant de représentation :

$$e \in E \Longleftrightarrow X[e] = true$$

Notes

– `elem` doit être un type discret.

– Pour les boucles **for** il faut disposer de son MIN et de son MAX (i.e. **type** `elem` = `MIN..MAX`).

– le résultat d’une fonction (ici `ens`) doit être un type non structuré en Pascal pur, mais la plupart des Pascal admettent cette extension.

4.1.1 Fonctions et procédures

```
type elem = MIN..MAX;
     ens = packed array[elem]of boolean;

function dans(e: elem; x: ens) : boolean; { O(1) }
begin
     dans := x[e];
end;
```

```
function ensVide : ens; { O(|elem|) }
var i   : elem;
    res : ens;
begin
     for i := MIN to MAX do res[i] := false;
     ensVide := res;
end;
```

```
function ajout(e: elem; x: ens): ens; { O(|elem|) }
var res: ens;
begin
     res := x;
     res[e] := true;
     ajout := res;
end;
```

```
function singleton(e: elem): ens; { O(|elem|) }
var i   : elem;
var res : ens;
begin
     for i := MIN to MAX do res[i]:= false;
     res[e] := true;
     singleton := res;
end;
```

```
function union(x,y:ens): ens; { O(|elem|) }
var i   : elem;    res : ens;
begin
     for i := MIN to MAX do res[i] := x[i] or y[i];
     union := res;
end;

function intersection(x,y:ens): ens; { O(|elem|) }
var i   : elem;    res : ens;
begin
     for i := MIN to MAX do res[i] := x[i] and y[i];
     intersection := res;
end;
```

```
procedure insérer(e: elem; var x: ens); { O(1) }
begin
     x[e] := true;
end;

procedure supprimer(e: elem; var x: ens); { O(1) }
begin
     x[e] := false;
end;
```

4.2 Liste

Le problème des vecteurs de bits est que l'on obtient une complexité en $\mathcal{O}(|elem|)$ (nombre de valeurs possibles) or on voudrait $\mathcal{O}(|x|)$ (nombre de valeurs utilisées).

Soit un TA liste, on peut s'en servir pour implémenter les ensembles.

idée

$$ajout \rightarrow cons$$
$$ensVide \rightarrow listeVide$$

Puisque ce sont des constructeurs similaires.

L'implémentation de $dans$ se déduit de ses équations :

```
function dans(e: elem; x: ens): boolean;
begin
     if null(x) then dans :=false
     else if head(x) = e then dans := true
          else dans := dans(e, tail(x));
end;
```

On obtient de même $intersection$ et $union$.

4.2.1 union

Exemple :

union(ajout(e, x), y) = ajout(e, union(x, y))

$\downarrow$

union(cons(e,x),y) = cons(e, union(x, y))

On remarque que les équations de $union$ sont exactement celles de $append$!
Donc $union \rightarrow append$.

4.2.2 intersection

```
function intersection(x,y: ens): ens;
var e : elem;
begin
    if null(x) then intersection := listeVide
    else
      begin
          e := head(x);
          if dans(e,y)
          then intersection := ajout(e, intersection(tail(x),y)
          else intersection := intersection(tail(x),y);
        end
end;
```

4.2.3 Temps de calcul

Si les listes sont implémentées par des listes chaînées, on a :

f	$T_f =$
listeVide	$\mathcal{O}(1)$
cons	$\mathcal{O}(1)$
append	$\mathcal{O}(\texttt{length}(x))$
head	$\mathcal{O}(1)$
tail	$\mathcal{O}(1)$

Les fonctions sur les ensembles prennent donc :

f	$T_f =$
ensVide	$\mathcal{O}(1)$
ajout	$\mathcal{O}(1)$
dans	$\mathcal{O}(\texttt{length}(x))$
singleton	$\mathcal{O}(1)$
union	$\mathcal{O}(\texttt{length}(x))$
intersection	$\mathcal{O}(\texttt{length}(x)*\texttt{length}(y))$!!!

4.2.4 Place en mémoire

De l'ordre de la longueur de la liste l.

4.2.4.1 Problème

Exemple : l'ensemble $x = \{1, 2\}$ peut être représenté par la liste $[1, 2, 1, 2, 1, 2, 1, 2]$.

Pour un ensemble x , la longueur de la liste n'est pas bornée !

4.3 Liste sans doubles

Pour que

$$length(x) \leq \mathcal{O}(|x|)$$

on pose comme invariant “pas de doubles” :

$$\forall i, j \in N_0^+ : i, j \leq length(x)\ et\ i \neq j \Rightarrow nth(i, x) \neq nth(j, x)$$

Il faut garantir que chaque fonction respecte bien le nouvel invariant.

– ensVide : OK.

– singleton : OK.

ajout : tester qu'on ne crée pas de double :

```
function ajout(e: elem; x: ens): ens;
begin
     if dans(e,x) then ajout := x
     else ajout := cons(e,x);
end;
```

Les autres fonctions se déduisent des axiomes.

4.3.1 Place en mémoire

de l'ordre du nombre d'éléments dans l'ensemble.

4.3.2 Temps d'exécution [Liste chaînée]

P	$T_P =$
ensVide	$\mathcal{O}(1)$
dans	$\mathcal{O}(\|x\|)$
ajout	$\mathcal{O}(\|x\|)$
singleton	$\mathcal{O}(1)$
union	$\mathcal{O}(\|x\| * \|y\|)$
intersection	$\mathcal{O}(\|x\| * \|y\|)$

Attention Apparemment, le temps augmente, mais pour les listes avec doubles, l n'est PAS bornée !

4.4 Listes triées

But diminuer le temps de $union$, $intersection$.

Idée éviter $dans \Rightarrow$ invariant : *La liste x est triée*

$$\forall i, j \in N : 0 < i < j < length(x) \Rightarrow nth(i, x) < nth(j, x)$$

Les anciennes opérations vérifient-elles le nouvel invariant ?

– ensVide : OK.

– singleton : OK.

ajout : insérer à la bonne place !

```
function ajout(e: elem; x: ens): ens;
begin
     if null(x) then ajout := cons(e, x)
     else if e  < head(x)
          then ajout := cons(e, x)
          else if e = head(x)
               then ajout := x
               else ajout := cons(head(x), ajout(e, tail(x));
end;
```

dans : OK mais peut aller plus vite :

```
function dans(e: elem; x: ens): boolean;
begin
     if null(x) then dans := false
     else if e < head(x) then dans := false
          else if e = head(x) then dans := true
               else dans := dans(e, tail(x));
end;
```

union : OK mais peut aller plus vite :

```
function union(x,y: ens): ens;
begin
  if null(x) then union := y
  else if null(y) then union:= x
    else if head(x) < head(y)
      then union := cons(head(x),union(tail(x),y))
      else if head(x) = head(y)
        then union := cons(head(x),union(tail(x),tail(y)))
        else union := cons(head(y),union(x, tail(y)));
end;
```

Intersection : même principe :

```
function intersection(x,y: ens): ens;
begin
  if null(x) then intersection := listeVide
  else if null(y) then intersection := listeVide
    else if head(x) < head(y)
      then intersection := intersection(tail(x),y)
      else if head(x) = head(y)
        then intersection :=
          cons(head(x),
               intersection(tail(x),tail(y)))
        else intersection := intersection(x, tail(y));
end;
```

4.4.1 Place en mémoire

La même : de l'ordre du nombre d'éléments.

4.4.2 Temps d'exécution

T(ensVide) = $\mathcal{O}(1)$

T(singleton) = $\mathcal{O}(1)$

T(ajout) = $\mathcal{O}(|x|)$

T(dans) = $\mathcal{O}(|x|)$

T(union) = $\mathcal{O}(|x| + |y|)$

T(intersection) = $\mathcal{O}(|x| + |y|)$

4.5 Tables de hachage

Idée

– Variante du vecteur de bits (Celui-ci utilise un tableau trop grand.)
– On diminue le type elem par une fonction qui le “compresse”.

```
function h(e: elem): indice
```

où elem est grand, mais indice est petit, par exemple $0..M$

P.ex. on peut prendre le reste de la division par $M + 1$.

4.5.0.1 Problème

Lorsque deux éléments distincts ont le même indice, il y a COLLISION.

4.5.0.2 Solutions

1. Représenter l'ensemble des éléments de même indice (p.ex. par une liste triée).
2. Représenter la liste dans la table : **Adressage ouvert**.
3. Agrandir la table de hachage.

4.5.1 Ensemble des collisions

```
type indice = 0..M;
     ens = array[indice] of ens2;
     {ens2 = par ex. liste chaînée}

function dans(e: elem; x: ens): boolean;
begin
     dans := dans2(e, x[h(e)])
end;
```

```
procedure inserer(e: elem; var x: ens);
begin
     inserer2(e, x[h(e)])
end;

procedure supprimer(e: elem;var x: ens);
begin
     supprimer2(e,x[h(e)])
end;
```

```
function ensVide : ens;
var i   : indice;
    res : ens;
begin
     for i := 0 to M do res[i] := ensVide2;
     ensVide := res;
end;
```

```
function ajout(e: elem; x: ens): ens;
var i   : indice;
    res : ens;
begin
     res := x ; { ! peut créer du partage}
     res[h(e)] := ajout2(e, x[h(e)]);
     ajout := res;
end;
```

```
function union(x,y: ens): ens;
var i   : indice;
    res : ens;
begin
     for i := 0 to M do res[i]:= union2(x[i],y[i]);
     union := res;
end;
```

4.5.1.1 Temps d'exécution

Opération	Temps au pire : $\mathcal{O}(\ldots)$	Temps moyen : $\mathcal{O}(\ldots)$
insérer	$T_{\texttt{insérer2}}(\lvert x\rvert)$	1
dans	$T_{\texttt{dans2}}(\lvert x\rvert)$	1
supprimer	$T_{\texttt{supprimer2}}(\lvert x\rvert)$	1
fusionner	M * $T_{\texttt{fusionner2}}(\lvert x\rvert)$	M
union	M * $T_{\texttt{union2}}(\lvert x\rvert)$	M
intersection	M * $T_{\texttt{intersection2}}(\lvert x\rvert)$	M
ensVide	M * $T_{\texttt{ensVide2}}(\lvert x\rvert)$	M

h est supposée en $\mathcal{O}(1)$

4.5.1.2 Place mémoire

$$\mathcal{O}(M) = \mathcal{O}(|x|) \text{ si M bien choisi.}$$

4.5.1.3 Conclusion

– Améliore `dans, insérer, supprimer`.
– Dégrade `ensVide, ajout`.

$\Rightarrow$ Souvent employé car $ensVide, ajout$ sont rares.

4.5.2 Représenter la liste des collisions dans la table

On met les éléments en collision à d'autres places libres du tableau

Avantage : pas de pointeurs, d'où économie de place.

On pourrait les mettre à la place suivante (si elle est libre) mais risque de collision en chaîne ou **cluster**.

double hachage : h_1, h_2 telle que $h_2(e)$ est premier par rapport à M+1 (ex. : M+1 = 2^n, h_2 impair). On suppose une valeur spéciale de $elem$: $vide$.

La suite des indices explorés est :

```
function h(e: elem; i: indice): indice;
begin
    h := (h1(e) + i * h2(e)) mod M+1
end;

{h(e,0), ..., h(e,M) est une permutation de 0..M}
```

Hypothèse : supprimer n'est pas dans le TA.

Invariant :

$$\exists j : e = X[j] \Rightarrow \forall i \in [0..k[\ ou\ h(e,k) = j, X[h(e,i)] \neq vide$$

Pour insérer, on parcourt la séquence d'indices jusqu'à trouver une case vide. Pour rechercher, on parcourt la liste d'indices jusqu'à trouver la clé ou tomber sur une case vide.

4.5.2.1 Temps

– Dépend du risque de “collision en chaîne”.
– Si négligeable, même temps moyen.

4.5.2.2 Place

Economie qui ne change pas l’ordre de grandeur :

$$\mathcal{O}(M) = \mathcal{O}(|x|)$$

4.5.3 Eviter les collisions en agrandissant la table

– Pas possible en Pascal car la taille des tableaux est fixe.

– Possible en C/C++.

– Nécessite une série de fonctions de hachage ; on prend souvent :

$h(e, n) \in 0..2^n - 1$ On suppose que pour n grand, h ne donne plus de collisions.

```
type ens = record
                n : integer; {>= n0 > 0}
                t : array [0..2^n −1] of elem        {pas du PASCAL}
           end;

procedure inserer(e: elem; var x: ens);
begin
     j := h(e, x.n);
     if x.t[j] = vide then x.t[j] := e
     else if x.t[j] <> e
          then begin
                    dedoubler(x);
                    insérer(e,x);
               end
```

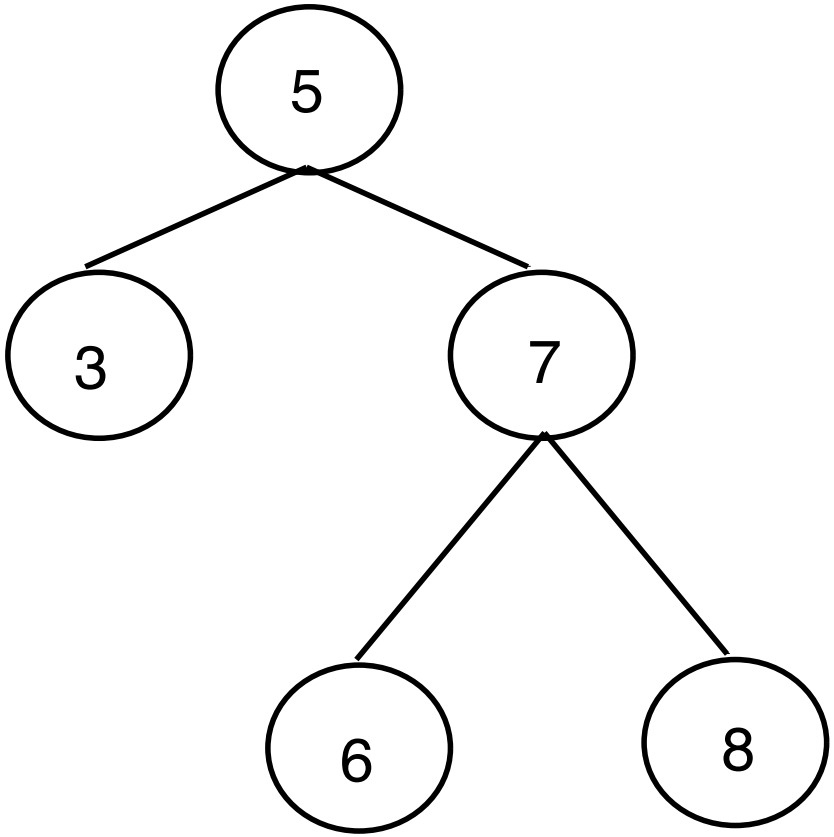
```
end;
```

4.6 Arbres binaires de recherche

Un arbre binaire est soit un arbre vide, soit constitué d'une valeur, et de deux sous-arbres : le fils de gauche et le fils de droite.

```
type arbre = arbreVide | cons(e: elem; g,d : arbre)
```

e est appelé la valeur de la racine de l'arbre. g et d, les fils gauche et droit.


5
3
7
6
8

```
arbre {binaire} = ^noeud;

noeud = record
          e : elem;
          g : arbre;
          d : arbre;
        end;
{inv: x.g < x,  x.d < x pour ‘‘<’’ bien fondé
      càd pas de cycles}
```

Un arbre binaire est **trié (ou de recherche)**, si les valeurs du sous-arbre de gauche sont plus petites que la racine, et celles du sous-arbre de droite sont plus grandes, et récursivement. La recherche (`dans`) y est plus efficace.

```
type abr {arbre binaire de recherche}
     = arbre
     {inv: si x <> nil:
               pour tout e1 dans Info(x^.g),
               pour tout e2 dans Info(x^.d),
                    e1 < x^.e < e2
               x^.g et x^.d sont des abr};
```

4.6.1 Fonctions et Procédures

```
function cons(e: elem; g,d : arbre) : arbre;
var p : arbre;
begin
    new(p);
    p^.e := e;
    p^.g := g;
    p^.d := d;
    cons := p;
end;
```

```
function ensVide : abr;
begin
     ensVide := nil
end;

function singleton(e: elem):abr;
begin
     singleton := cons(e,ensVide,ensVide)
end;
```

```
function ajout(e: elem; x: abr):abr;
begin
  if x = nil then ajout := singleton(e)
  else if e < x^.e
    then ajout := cons(x^.e, ajout(e, x^.g), x^.d)
    else if e > x^.e
      then ajout := cons(x^.e, x^.g, ajout(e, x^.d))
      else  ajout := x
end;
```

4.6.2 Recherche

```
function dans(e: elem; x: abr): boolean;
begin
     if x = nil then dans := false
     else if e < x^.e then dans := dans(e,x^.g)
          else if e > x^.e then dans := dans(e, x^.d)
               else dans := true
end;
```

4.6.2.1 Elimination de la récursivité terminale

```
function dans2(e: elem; x: abr): boolean;
var trouve : boolean;
begin
     trouve := false;
     while (x <> nil) and not trouve do
           if e < x^.e then x := x^.g
           else if e > x^.e then x:= x^.d
                else trouve := true;
     dans2:= trouve
end;
```

4.6.3 Insertion

```
procedure inserer(e: elem; var a: abr);
begin
  if a = nil then a := singleton(e)
  else if e < a^.e then inserer(e, a^.g)
    else if e > a^.e then inserer(e, a^.d)
  {if e = a^.e : il s'y trouve déjà, ne rien faire}
end;
```

4.6.4 Supprimer

1. Si le noeud à supprimer n'a pas de fils gauche, on peut l'enlever
2. Sinon, il nous manque une valeur pour séparer les deux sous-arbres : on remonte le max du fils gauche.

On pourrait mettre un sous-arbre sous l'autre, mais ça déséquilibrerait l'arbre

```
procedure supprimer(e: elem; var x: abr);
begin
     if x <> nil then
        if e < x^.e then supprimer(e,x^.g)
        else if e > x^.e then supprimer(e, x^.d)
             else SupprimerRacine(x)
end;
```

```
procedure SupprimerMax(var a: abr; var m: elem);
(*  Pré: a <> nil
Post : m = Max(Info(a0)) et Info(a) = Info(a0) \ {m}  *)
var p : abr;
begin
     if a^.d = nil
     then begin
               m := a^.e;
               p := a;
               a :=a^.g;
               dispose(p);
          end
     else SupprimerMax(a^.d,m)
end;
```

```
procedure SupprimerRacine(var a: abr);
(*  Pré: a <> nil
Post : Info(a) = Info(a0) \ {a0^.e}  *)
var p : abr;
begin
  p := a;
  if a^.g = nil
  then begin
    a := a^.d;
    dispose(p);
  end
  else SupprimerMax(a^.g,a^.e)
end;
```

4.6.4.1 Temps d'exécution

Le temps est de l'ordre de la hauteur, mais au pire elle peut être n.

Opération	Temps au pire : $\mathcal{O}(\ldots)$	Temps moyen : $\mathcal{O}(\ldots)$
dans	n	$log(n)$
insérer	n	$log(n)$
supprimer	n	$log(n)$
ensVide	1	1
ajout	n	$log(n)$

Il faut équilibrer l'arbre.

4.7 Arbres rouges/noirs

4.7.1 Définition

On ajoute une “couleur” binaire à chaque noeud, et surtout des invariants de données pour que la hauteur soit logarithmique :

– le nombre de noeuds noirs est le même sur toute branche
– le fils d’un noeud rouge est noir.

Les noeuds rouges donnent un peu de souplesse à la contrainte d’équilibre.

Note : on pourrait aussi choisir la couleur de la racine.

4.7.1.1 Déclaration Pascal

```
rougenoir = ^noeud;

noeud = record
            e : elem;
            g : rougenoir;
            d : rougenoir;
            rouge : boolean;
        end;
```

4.7.1.2 Invariants de données

- l'arbre est acyclique : `h(x^.g) < h(x)` et `h(x^.d) < h(x)`
- l'arbre est trié : si `x <> nil`, pour tout `e1` dans `Info(x^.g)`, `e1 < x^.e` pour tout `e2` dans `Info(x^.d)`, `x^.e < e2`
- le fils d'un rouge est noir : si `rouge(x)` alors non `rouge(x^.g)` et non `rouge(x^.d)`
- toute branche contient le même nombre de noeuds noirs : `hn(x^.g) = hn(x^.d)`
- l'invariant est récursif : `x^.g` et `x^.d` sont des rougenoir

La **hauteur noire** d'un arbre rouge/noir est le nombre de noeuds noirs rencontrés le long d'une branche :

```
function hn(a: rougenoir): integer;
begin
  if a=nil then hn:=0
  else if a^.rouge then hn := hn(a^.d)
       else hn := 1+hn(a^.d)
end
```

Note : dans la nouvelle édition de Cormen, `hn` est noté `bh` (black height)

Exercice : Démontrez que pour tout arbre rouge/noir :

1. $hn \leq h \leq 2 * hn + 1$
2. $2^{hn} - 1 \leq n \leq 2^h - 1$
3. $h = \mathcal{O}(\log n)$

4.7.2 Fonctions de base

On adapte les fonctions des arbres binaires triés, pour qu'elles traitent les arbres rouges/noirs :

`cons` doit recevoir la couleur. La précondition doit être plus forte :

- `hn(g)` = `hn(d)`
- c = noir ou (c = rouge et non rouge(g) et non rouge(d))
- $\forall x \in Info(g).x < e$
- $\forall x \in Info(d).x > e$

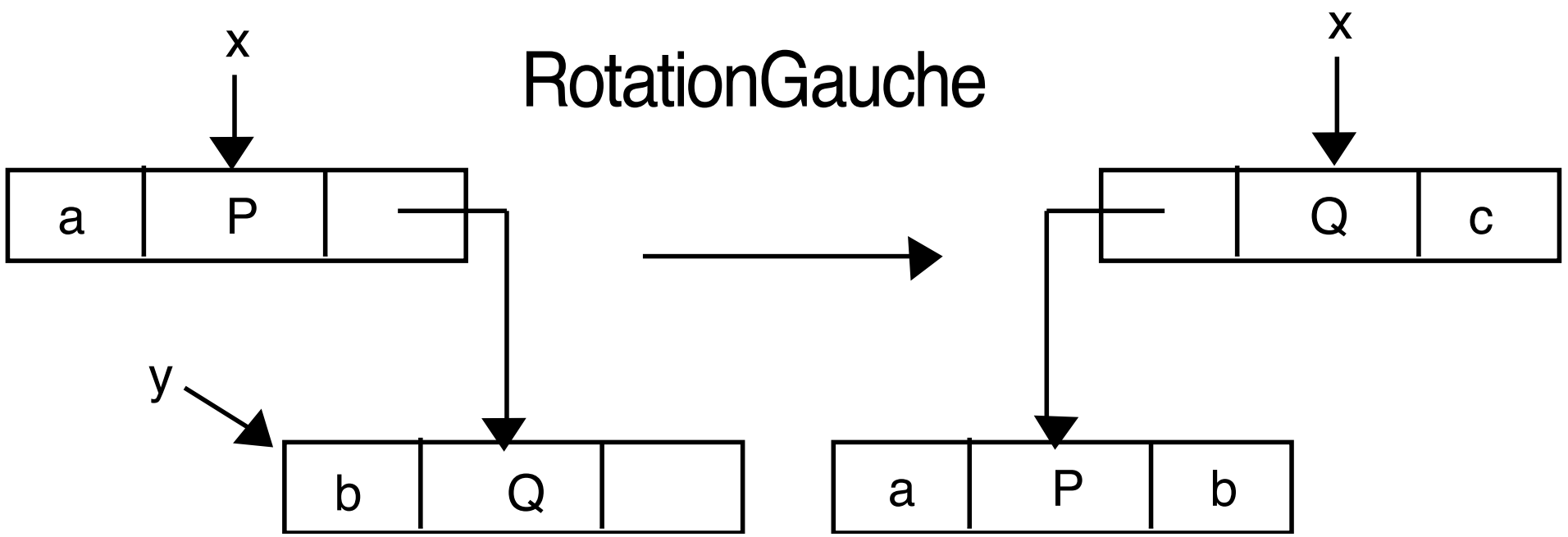
```
function cons(e: elem;g,d : rougenoir; c: boolean): rougenoir;
var p : rougenoir;
begin
     new(p);
     p^.e := e;
     p^.g := g;
     p^.d := d;
     p^.rouge := c;
     cons := p;
end;
```

Sans aucun changement, la recherche s'exécute en temps logarithmique.

```
function dans(e: elem; x: rougenoir): boolean;
begin
  if x = nil then dans := false
  else if e < x^.e then dans := dans(e, x^.g)
    else if e > x^.e then dans := dans(e, x^.d)
      else dans := true
end;
```

4.7.3 Rotations

Nous voulons que Insérer et Supprimer s'exécutent en temps $\log n$. Nous recherchons donc des opérations pour rééquilibrer l'arbre à faible coût.

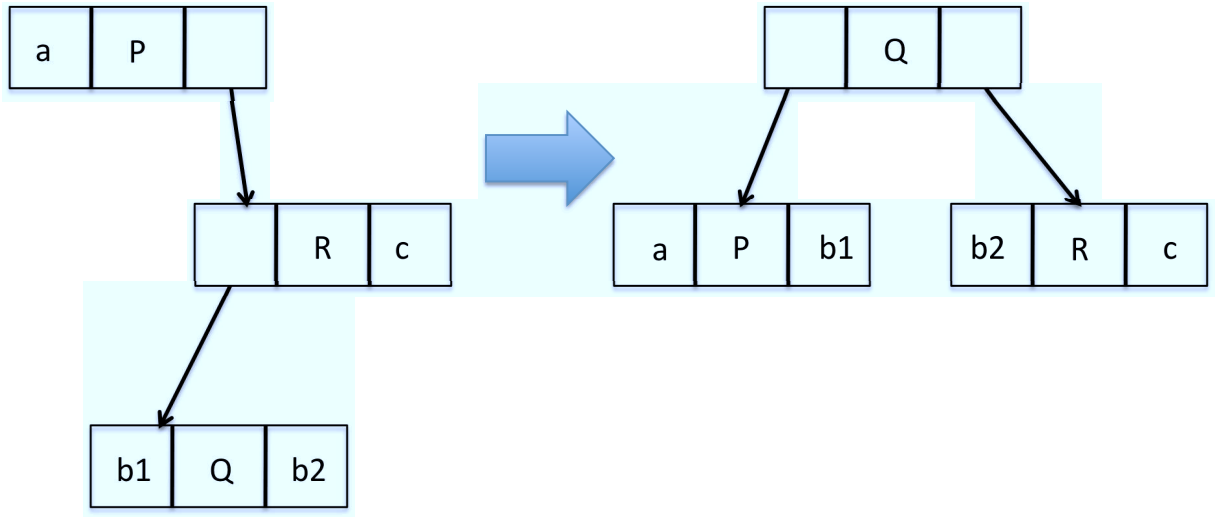


Une rotation gauche prend un temps constant ; elle raccourcit les branches de `c` et rallonge celles de `a`.

L'arbre reste trié mais pas toujours rouge-noir.

```
procedure RotationGauche(x: abr);
var y : abr;
begin
     y := x^.d; {<> nil}
     echanger(x^.e,y^.e);
     x^.d := y^.d;
     y^.d := y^.g;
     y^.g := x^.g;
     x^.g := y;
end;
```

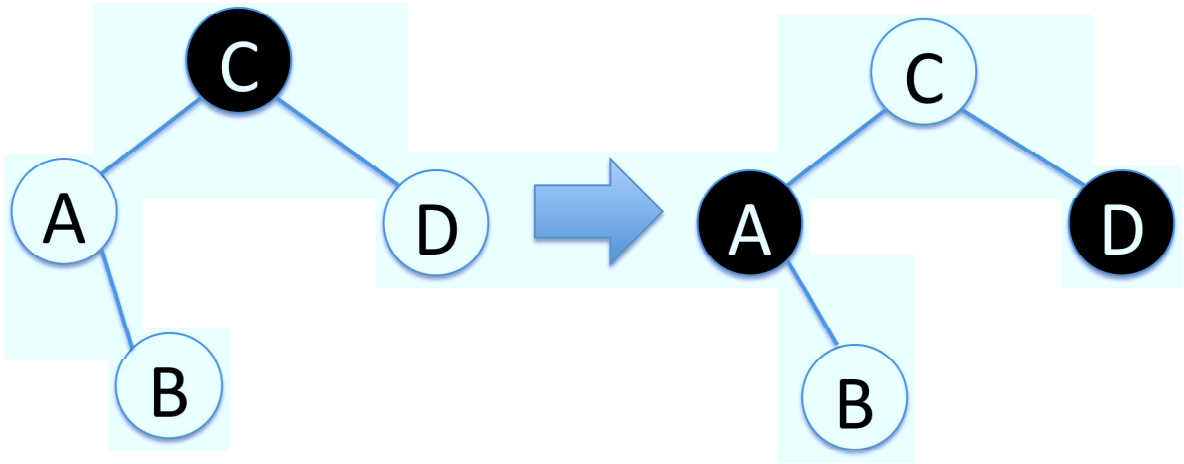
Si c'est `b` qu'il faut raccourcir, on fait d'abord une rotation droite sur `y`.



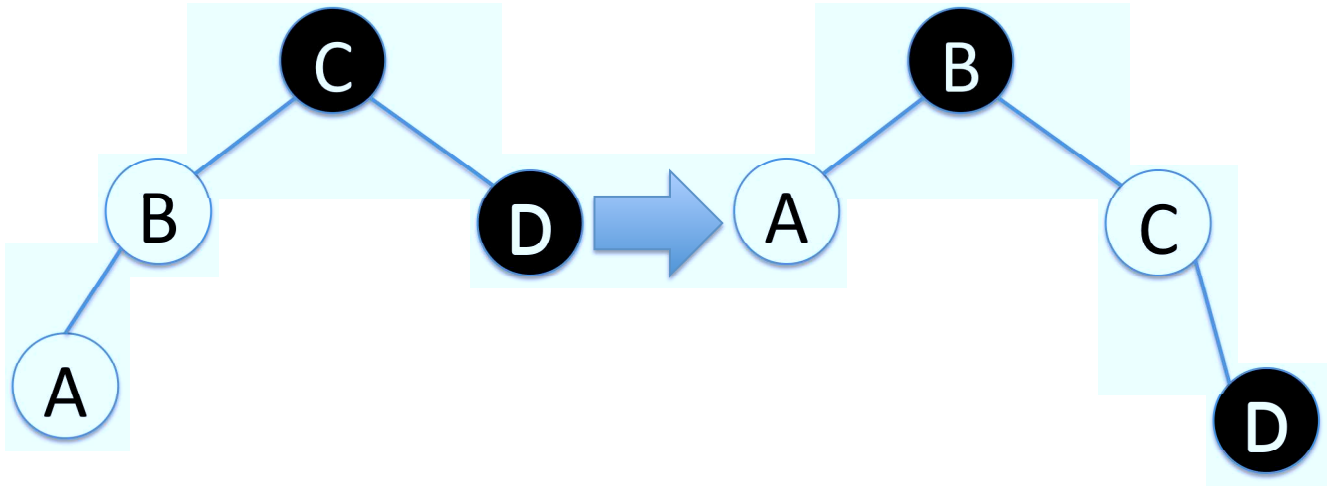
4.7.4 Insertion

1. On fait une insertion dans un ABR d'un noeud rouge
2. Si le père p est rouge : il faut rétablir l'invariant

Cas 1 : si l'oncle est rouge : le problème est remonté sur le grand-père.

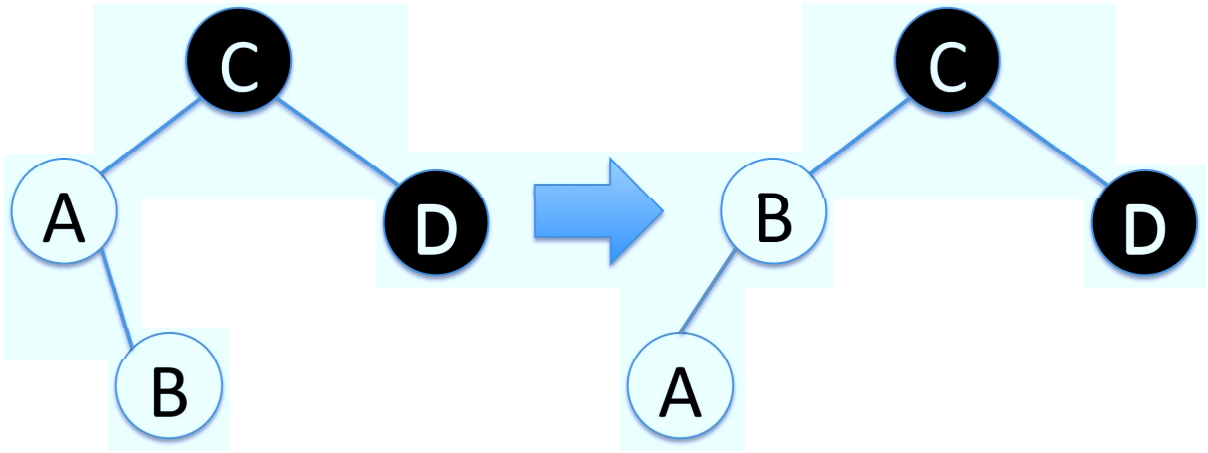

C
A
D
B
C
A
D
B

Cas 3 : si l'oncle est noir et le fils est du même côté que son père : on fait une rotation qui résoud le problème.



Cas 3

Cas 2 : si l'oncle est noir et le fils est de l'autre côté que son père : on fait d'abord une rotation qui ramène au cas 3.



Cas 3

```
procedure insererec(e: elem; var a,b,c : rougenoir);
{cas général de inserer
Pré: a,b <> nil
     b est le fils de a où e doit être inséré.
     c est le fils de b où e doit être inséré.
Post: a = a0
      b = b0
      Info(c)= Info(c0) u {e}}
var y : rougenoir; {oncle}
begin
  if c = nil  then c := singleton(e)
  else if e < c^.e then insererec(e,b,c,c^.g)
    else if e > c^.e then insererec(e,b,c,c^.d);
```

```
if b^.rouge and c^.rouge then {noir(a)}
 if b = a^.g
  then begin
    y := a^.d;
    if rouge(y)
    then begin {Cas 1}
     b^.rouge := false;
      y^.rouge := false; {<> nil car rouge(y)}
      a^.rouge := true;
    end
    else begin {noir(y)}
      if c = b^.d then RotationGauche(b) {Cas 2};
     RotationDroite(a) {Cas 3};
   end
```

end

Plus les cas symétriques.

```
procedure inserer(e: elem; var a: rougenoir);
{a lancer sur la racine de l'arbre, cette procedure traite
les cas de niveau < 2 et envoie le reste à insererec}
begin
  if a = nil then a := singleton(e)
  else begin
    a^.rouge := false;
    if e < a^.e
    then if a^.g = nil
      then a^.g := singleton(e)
      else if e < a^.g^.e
        then insererec(e,a,a^.g,a^.g^.g)
        else if e > a^.g^.e
          then insererec(e,a,a^.g,a^.g^.d)
```

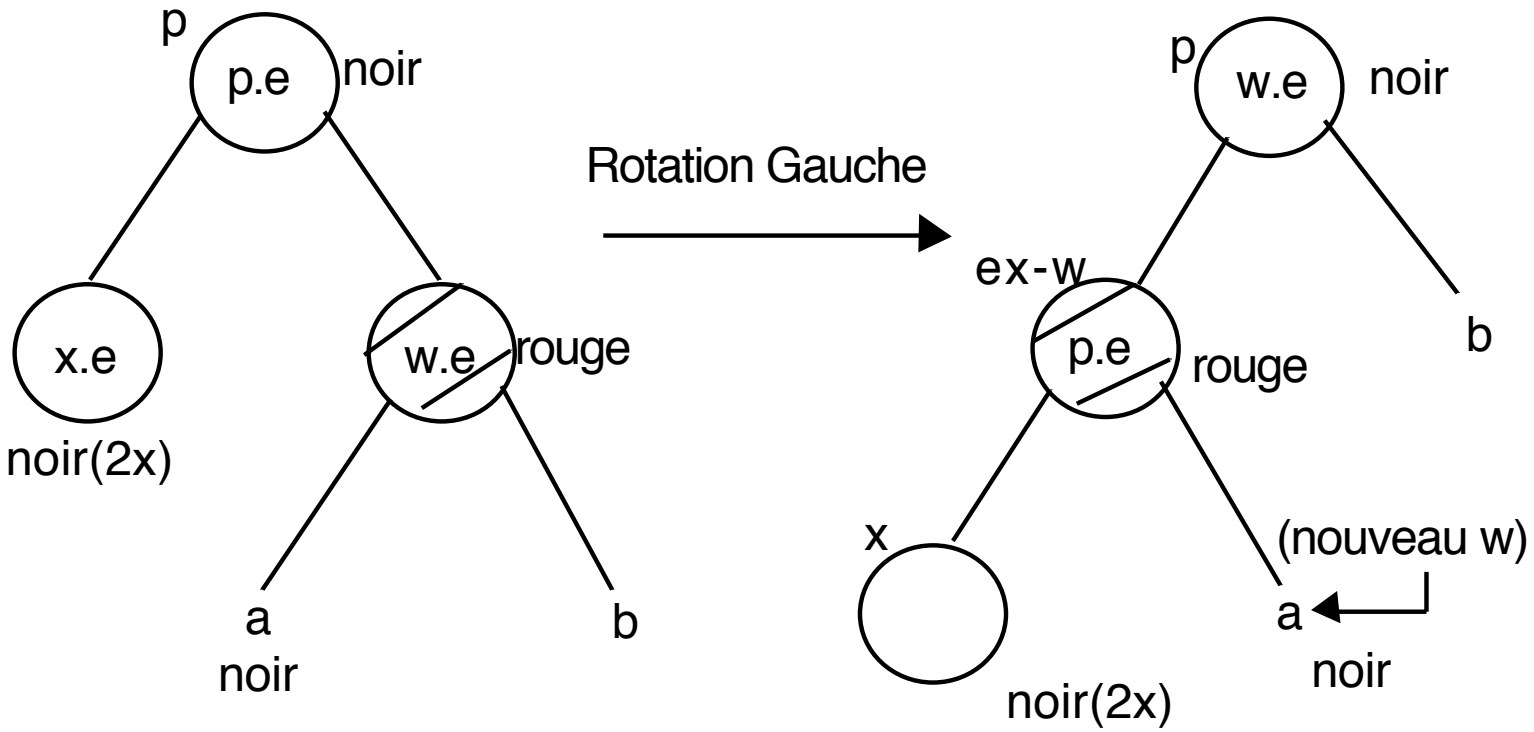
```
            else {rien à faire}
        else if e > a^.e
          then if a^.d = nil
            then a^.d := singleton(e)
            else if e < a^.d^.e
              then insererec(e,a,a^.d,a^.d^.g)
              else if e > a^.d^.e
                then insererec(e,a,a^.d,a^.d^.d)
                else {rien à faire}
    end
end;
```

4.7.5 Supprimer

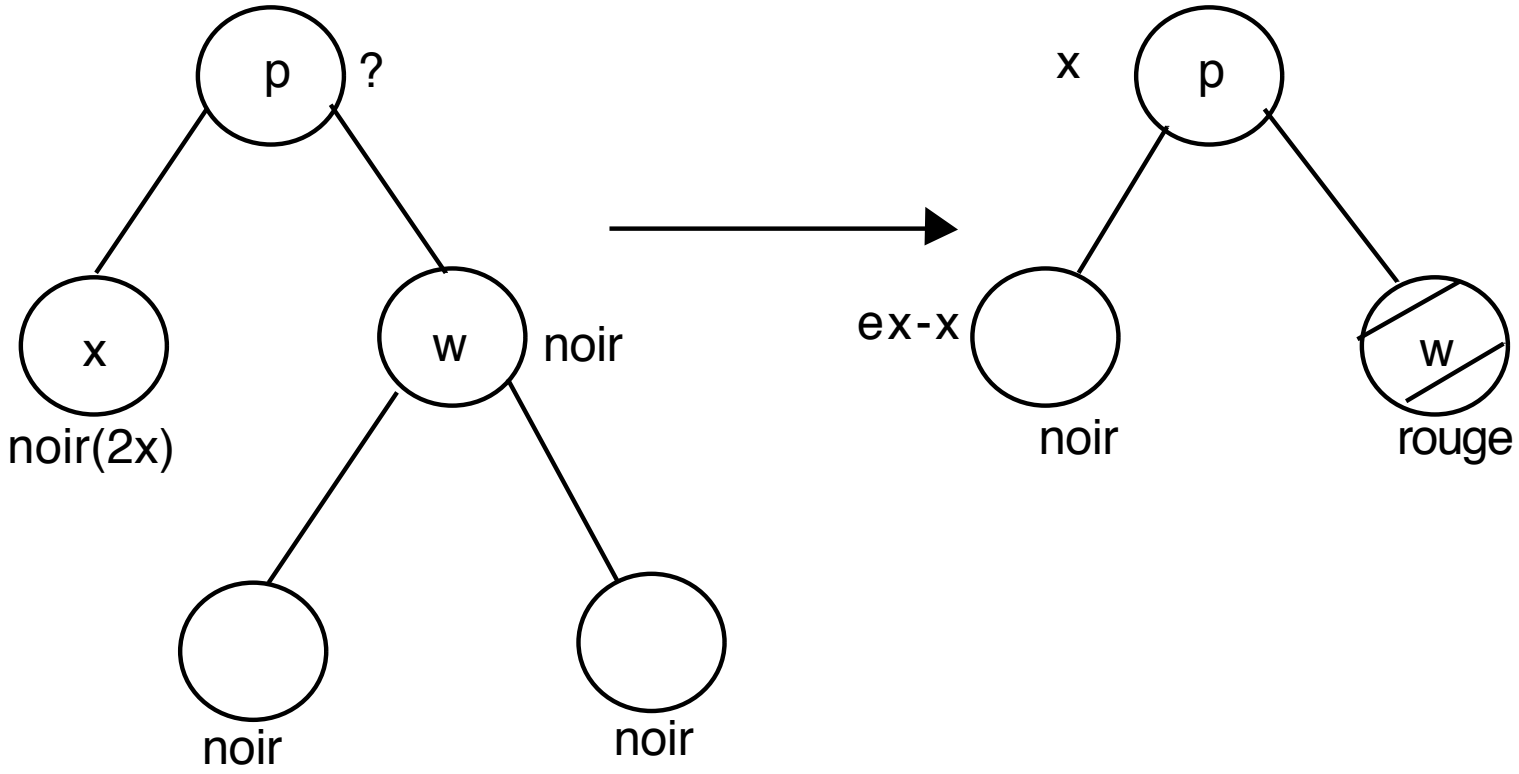
1. On retire comme dans un arbre binaire trié ;
2. Si le noeud retiré est rouge : OK ;
3. sinon on appele `retablir(x)` : il manque un noir sur tous les chemins qui passent par `x` (bulle). `x` est le fils du noeud retiré, qui le remplace.

4.7.5.1 Rétablir

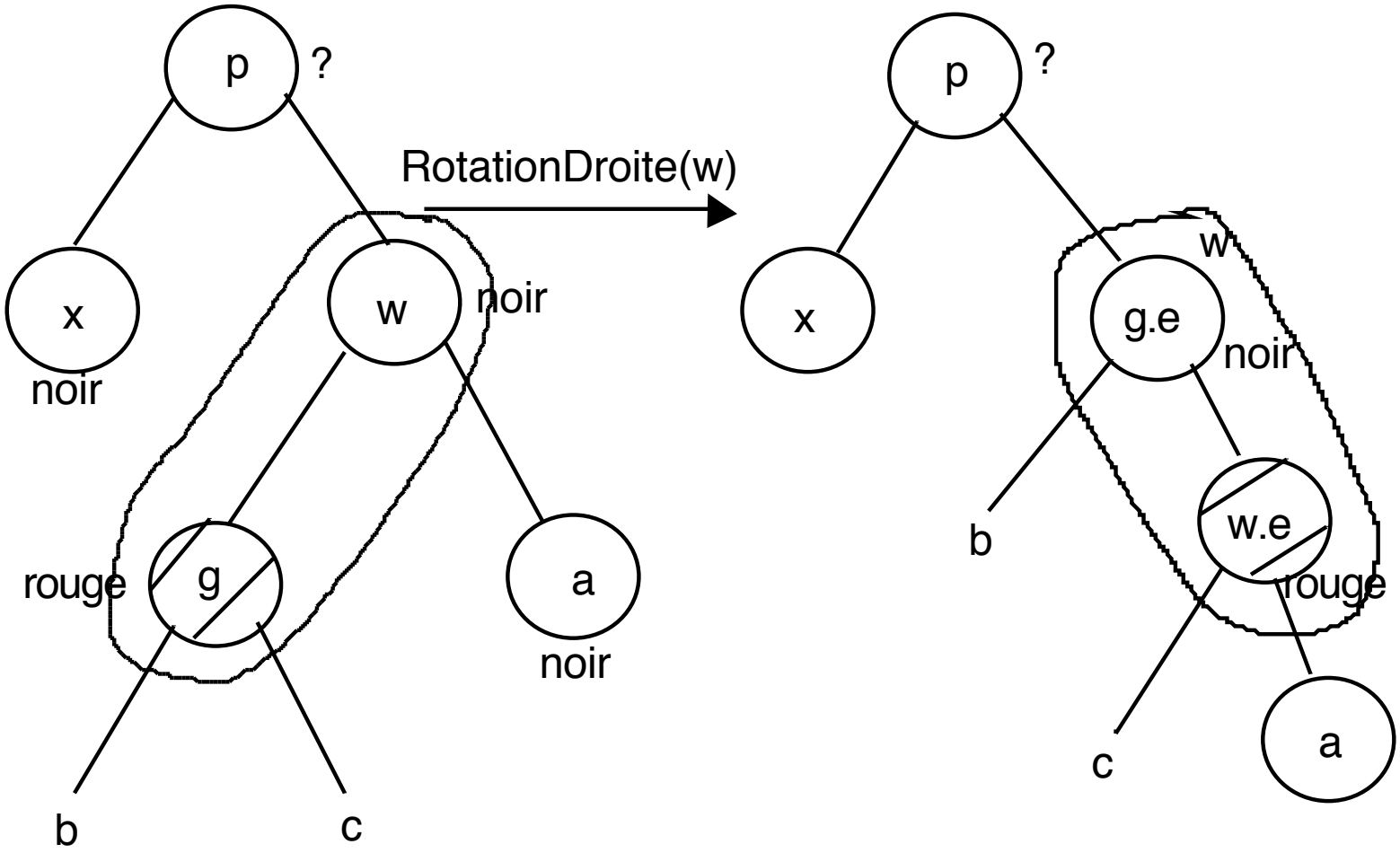
Si x est rouge, on le met à noir et la bulle est résolue. SI w, le frère de x, est rouge, on fait une rotation : le nouveau frère est noir.



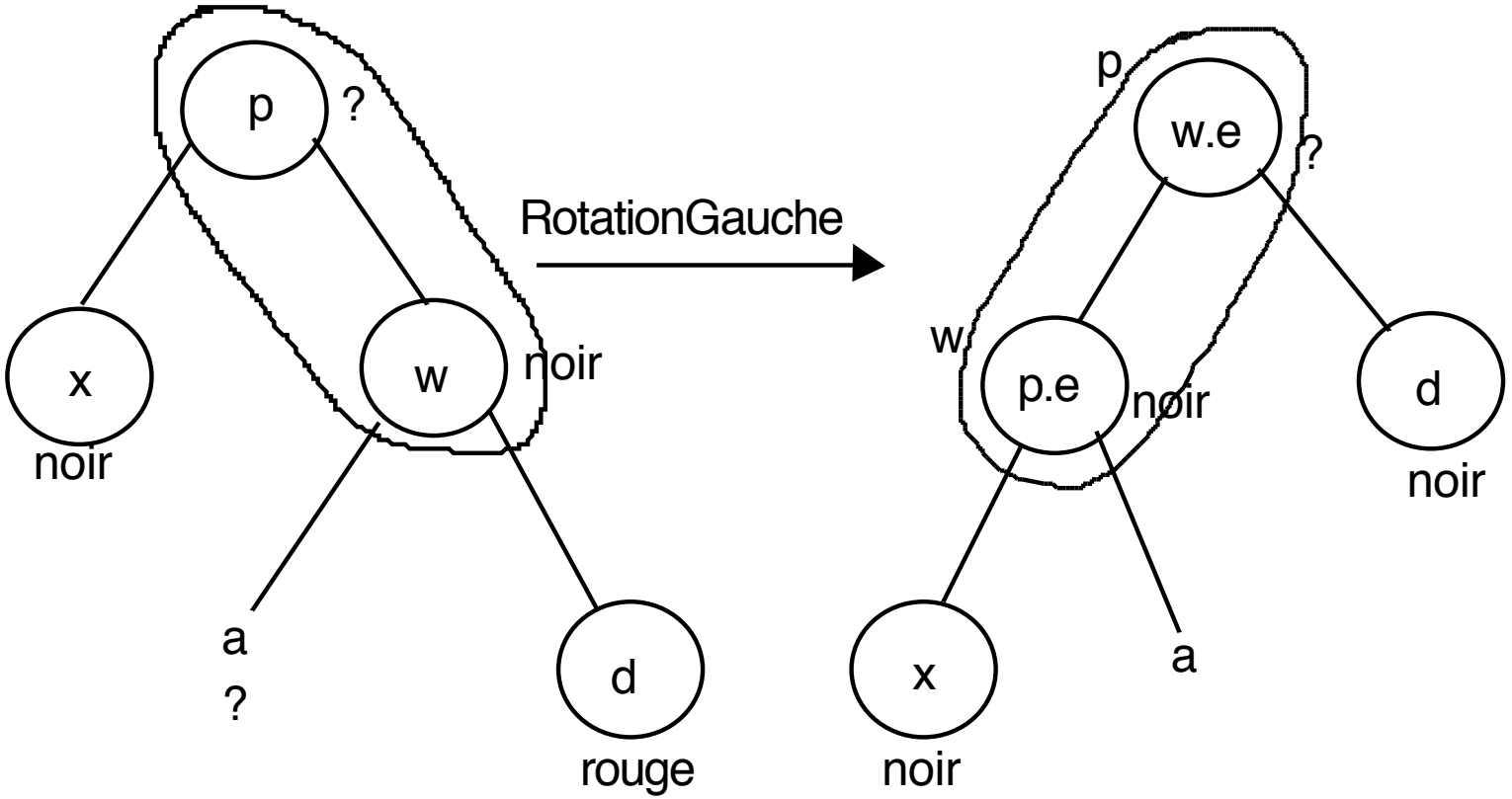
Si les neveux de x sont noirs, on rougit w, ce qui remonte la bulle sur p : appel récursif. (Si p était rouge, on rétablit l'invariant en le mettant à noir).



SINON SI le neveu de gauche est rouge et le neveu de droite est noir : On fait une rotation, qui rend le neveu de droite rouge.



Enfin une rotation sur p élimine la bulle en x ; on noircit d.



4.8 B-Arbres

Les B-arbres utilisent la même idée que les arbres rouges-noirs. Ils regroupent un paquet de noeuds bicolores de niveau proches dans un noeud de B-arbres.

1. Si on regroupe un noeud noir avec ses éventuels fils rouges, on obtient un arbre où toutes les branches ont la même longueur mais dont le nombre de fils varie entre 2 et 4.
2. Dans une feuille, tous les fils sont vides, on peut donc supprimer les pointeurs pour mettre plus de valeurs.
3. En général, on varie le nombre de fils entre t et $2.t$.
4. Dans un noeud, on peut mettre les valeurs dans un tableau de taille constante.

4.8.1 Définition

On choisit

```
const {pour un noeud intérieur, non feuille}
      Minelems = ...; {> 0: nombre min de clefs}
      Maxelems = ...; {> 2*Minelems : nombre max de clefs }
  {pour une feuille:}
      Mindonnees= ...; {> 0: nombre min d'éléments}
      Maxdonnees = ...; {> 2*Mindonnees: nombre max d'éléments}
```

Le nombre minimum de fils t est Minelems+1.

Soit Min = Mindonnees pour une feuille, Minelems sinon. La cellule racine peut aller en dessous du Min, mais doit avoir au moins 1 clef et 2 fils.

4.8.1.1 Déclaration

```
Barbre = ^noeud;
        {invariant : b <> nil}
 noeud = record
                uti : integer;  {nombre d'éléments utilisés}
                case feuille:boolean of
                     true: (donnees: array[1..Maxdonnees] of elem);
                     false: (elems: array[1..Maxelems] of elem;
                             fils: array[0..Maxelems] of Barbre);
                end;
          end;
```

4.8.1.2 Invariant de données

1. uti >= 0
2. uti > 0 si la racine n'est pas une feuille
3. uti <= Max
4. pour tout noeud sauf la racine, uti >= Min
5. pour tout i in 1..uti, e in Info(fils[i-1]), e < elems[i]
6. pour tout i in 1..uti, e in Info(fils[i]), elems[i] < e
7. tous les fils ont la même hauteur.
8. la partie utilisée de donnees et elems est triée
9. l'invariant est aussi vrai pour chacun des fils

L'invariant nous donne une hauteur logarithmique : la racine a au min une clé et deux fils, ses fils ont au minimum t fils, etc.

$$n \geq 1 + (t-1).\sum_{i=1}^{h} 2t^{i-1}$$

$$h \leq \log_t(\frac{n+1}{2})$$

4.8.2 Procédures et fonctions

4.8.2.1 Recherche

On recherche la clef dans le tableau `elems` ; Soit on l'y trouve, sinon on trouve le fils dans lequel il pourrait se trouver. Cette recherche peut être linéaire, ou dichotomique si Max est grand.

4.8.2.2 Ensemble vide

L'ensemble vide est représenté par une feuille avec uti=0.

4.8.2.3 Diviser

Un noeud plein peut être divisé en deux noeuds, et la clé centrale est remontée dans le père pour séparer ces deux noeuds.

1. Technique préventive : On met dans la précondition que le père n'est pas plein.
2. Technique corrective : On divise le père après si nécessaire par un appel récursif.

4.8.2.4 Insérer

On voudrait insérer une nouvelle clef dans la feuille où on devrait la trouver, mais ceci peut faire déborder la feuille si elle était pleine.

1. Technique préventive : on divise tous les noeuds pleins qu'on traverse
2. Technique corrective : on divise en remontant.

```
procedure inserec(e: elem; var b: Barbre);
   { Pré: b^ est non plein
     Post: b est un B–arbre; e s'ajoute à son contenu   }
  var i : integer;
  begin
       i := rech(e,b);
       if b^.feuille
       then if absent(e,i,b) then insererpos(i,e,b)
       else if absent(e,i,b)
            then if plein(b^.fils[i])
                 then begin
                      diviser(b,i);
                      if b^.elems[i+1] <= e then i := i + 1;
                      if b^.elems[i] <> e then inserec(e,b^.fils[i]);
```

```
                    end
                    else inserec(e,b^.fils[i])
    end;
```

```
begin {de inserer}
      if plein(b)
      then begin
          new(p);
          p^.feuille := false;
          p^.uti := 0;
          p^.fils[0] := b;
          b := p;
          diviser(p,0);
      end;
      inserec(e,b);
end;
```

4.8.2.5 Fusion

Symétriquement, la fusion supprime une clef du père qui pourrait ainsi passer en dessous du minimum.

4.8.2.6 Supprimer

Technique préventive : on ne descend dans un fils que s'il est au-dessus du min. Pour garantir cela, soit on prend des fils d'un frère (s'il n'est pas au min), soit on fusionne avec un frère.

De même lors de la suppression du minimum/maximum ci-dessous.

Si la valeur à supprimer est dans un noeud intérieur, il faut la remplacer soit :

1. par la valeur suivante : le minimum du fils de droite
2. par la valeur précédente : le maximum du fils de gauche

Si ces deux fils sont au min, on les fusionne avant.

Question de départ

Comment passer d'un **problème** décrit par une spécification

↓ *Architecture + Conception d'algorithme*

un **algorithme** = une idée de solution, indépendante du langage

↓ *Codage*

un **programme**

↓ *Compilation*

pour finalement obtenir un **exécutable**

Dans ce cours nous nous limitons à la conception d'algorithmes.

Méthodes Générales de Conception d'Algorithmes

1. Générer et tester
2. Diviser pour règner
3. Programmation dynamique
4. Algorithmes gloutons
5. . . .

CHAPITRE 5

DIVISER POUR RÈGNER

5.1 Idée

Utiliser la récursion, ce qui implique de diviser un problème en sous-problèmes similaires mais plus simples.

– Diviser le problème en n sous-problèmes similaires plus simples (typiquement : $n = 2$).
– Ces sous-problèmes sont résolus récursivement jusqu'aux "cas de base".
– Les solutions sont recombinées pour donner une solution du problème originel.

5.1.1 Cas de base

La notion de “plus simple” doit être une relation bien fondée.

Les cas de base sont les cas minimaux de cette relation ;
Pour $<$ sur $\mathbb{N}$, c’est le cas 0.

La fonction *résoudre*(d) est donc de la forme

Si cas de base(d)	Alors r := *résoudre-directement(d)*
Sinon	$(d_1, d_2, \ldots, d_n)$:= *diviser*(d)
	pour chaque i : r_i := *résoudre*(d_i)
	r := *combiner*($r_1, r_2, \ldots, r_n$)

5.1.2 Choix possibles

On peut choisir *diviser* ou *combiner* d'où on déduit l'autre.

Par exemple, on peut diviser un problème :

1. Suivant la définition récursive des données.
   Exemple : puisque liste = listevide | cons(elem,liste) $\Rightarrow$ division en tête et reste.
   Cette méthode à l'avantage d'être simple, même si elle est parfois déséquilibrée et donc moins efficace.
2. Pour avoir des sous-problèmes de même taille (souvent plus efficace).
   Exemple : deux listes ayant environ la même longueur.
   Dans ce cas, on doit aussi traiter le cas 1 comme cas de base.

5.2 Exemple : le tri : Spécification

tri :

En entrée : d : liste

En sortie : o : liste

Postcondition : o est triée et d est une permutation de o.

Si on veut qu'une solution existe toujours, on suppose que l'ordre de tri est réflexif ce qui permet les doublons.

5.2.1 Solution D1 : Tri par INSERTION

$diviser(l)$ = (tête(l), reste(l))

$\Rightarrow$ case de base : l = nil.

On déduit $combiner$ = insérer un élément dans une liste triée. (en $\mathcal{O}(n)$)

Le temps total d'exécution sera donc de l'ordre de $\mathcal{O}(n^2)$.

ins(e,nil) = cons(e,nil)

ins(e,cons(t,r)) = if e leq t then cons(e,cons(t,r))

else cons(t, ins(e, r))

5.2.2 Solution D2 : Tri par FUSION

$diviser(l)$ = (l_1, l_2) tel que $l_1 + l_2 = l$ et $|l_1| = |l_2| \pm 1$. (en $\mathcal{O}(\log n)$)

$\Rightarrow$ case de base : $|l| \leq 1$.

On déduit $combiner$ = fusionner deux listes triées. (en $\mathcal{O}(n)$).

Le temps total d'exécution sera donc de l'ordre de $\mathcal{O}(n \log n)$.

5.2.3 Solution C1 : Tri par SELECTION

$combiner$ = cons(e,l).

$\Rightarrow$ cas de base : l = nil.
On déduit $diviser$ = trouver le minimum des éléments de l. (en $\mathcal{O}(n)$)
Le temps total d'exécution sera donc de l'ordre de $\mathcal{O}(n^2)$, car on fait n appels à $diviser$.

5.2.4 Solution C2 : $\approx$ QUICKSORT

$combiner$ = append(l_1, l_2)

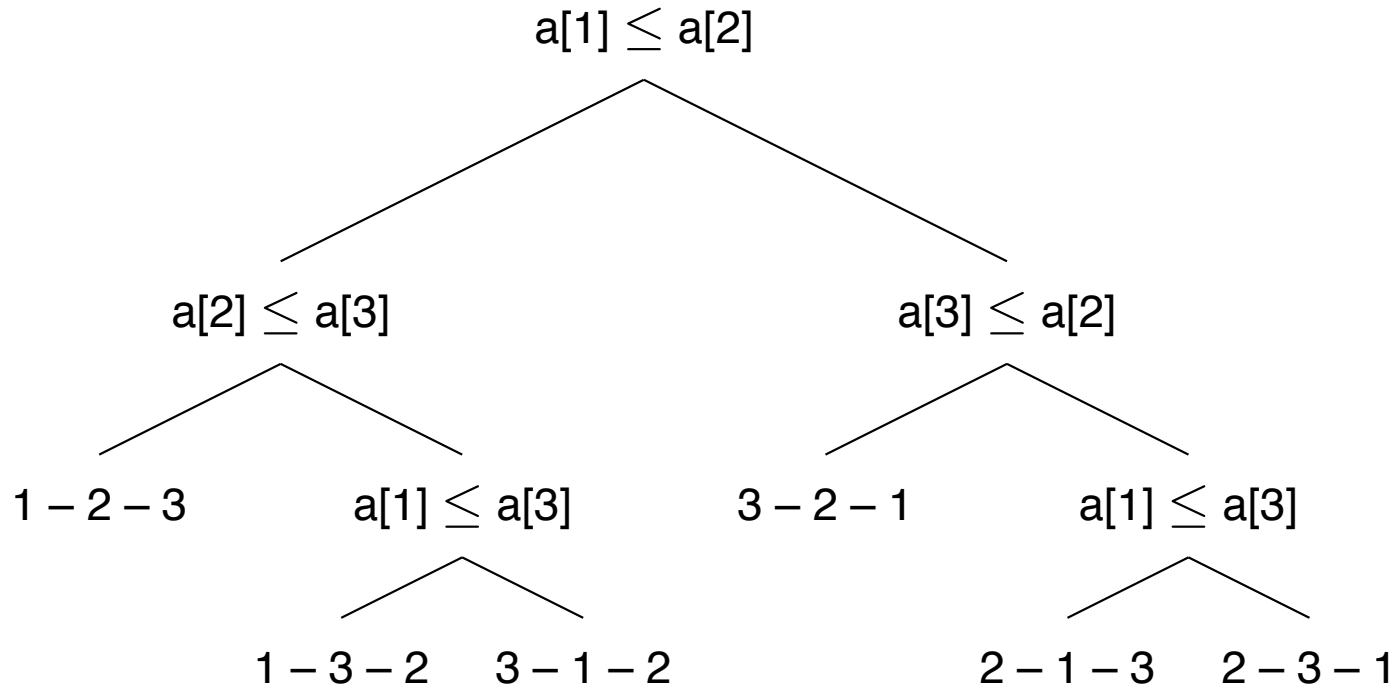
$\Rightarrow$ cas de base : $|l| \leq 1$.

On déduit $diviser$ = trouver la médiane de l, puis diviser en plus grand, plus petit autour de cette valeur pivot.

(variante : prendre un élément au hasard $\rightarrow$ QUICKSORT (en $\mathcal{O}(n \log n)$ si on ne fait pas toujours le mauvais choix)).

5.2.5 Temps d'exécution minimal d'un tri

Si le tri se fait par comparaisons ($\leq$), on peut représenter les éxecutions possibles par un arbre de décisions binaires : si le test est vrai, on va à gauche ; sinon, à droite.



Le nombre de feuilles est au moins $n!$, car il y a $n!$ permutations possibles de la

liste d'entrée.

Le temps d'exécution au pire T (en ne comptant que les comparaisons) est la hauteur de l'arbre d'exécutions.

Un arbre binaire de hauteur T a au plus 2^T feuilles.

$$n! \leq 2^T \iff \log_2(n!) \leq T$$

Or $\log(n!) = \mathcal{O}(n \log n)$ par l'approximation de Stirling.

D'où $T \geq \mathcal{O}(n \log n)$

$\mathcal{O}(n \log n)$ = meilleur ordre de grandeur possible pour un tri par comparaisons.

5.3 Multiplication

5.3.1 Méthode classique

Cette méthode est dérivée du calcul écrit classique.

1101
$\times$ 101
=
1101
+11 01
=
100 0001

Soit n le nombre de bits : on obtient une complexité en $\mathcal{O}(n^2)$ car on effectue n additions de n bits.

5.3.2 Diviser pour règner

On peut toujours augmenter n à une puissance de 2. Coupons X, Y en 2 parties égales :

$$X = A2^{n/2} + B \quad (1)$$

$$Y = C2^{n/2} + D \quad (2)$$

(3)

$$X * Y = (A * C)2^n + (A * D + B * C)2^{n/2} + (B * D)$$

qui est composé de 4 multiplications et dont le temps d'exécution suit

$$T(1) = 1$$
$$T(n) = 4T(n/2) + cn$$
$$\Rightarrow T(n) = \mathcal{O}(n^2)$$

Même temps par diviser pour règner !

Ceci est dû aux 4 appels récursifs à la multiplication. La première et la dernière multiplication calculent le début et la fin du résultat et sont donc inévitables : donc on s'attaque au terme central, en supposant le premier et le dernier connus.

On obtient alors 3 multiplications par la décomposition

$$X * Y = (A * C)2^n + ((A - B) * (D - C) + A * C + B * D)2^{n/2} + B * D$$

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/2) + cn \\ \Rightarrow T(n) &\leq \mathcal{O}(n^{\log_2 3}) \end{aligned}$$

Cette méthode a été trouvée par Karatsuba et Ofman.

5.3.2.1 Puissances rapides

La $n^{\text{ième}}$ puissance d'un nombre r est $r^n = r.r.r.....r$. Une implémentation évidente est de faire une boucle **for**, avec un temps $\mathcal{O}(n.m)$ où m est le temps d'une multiplication.

Par diviser pour règner, on a l'idée de couper la liste de r en deux parties égales.

```
function puissance(r: real; n: natural): real;
begin
  if n=0 then puissance := 1
  else if pair(n)
       then puissance := sqr(puissance(r,n div 2))
       else puissance := puissance(r,n−1)∗r
end
```

$$T = \mathcal{O}(m.logn)$$

5.3.2.2 Fibonacci par puissances rapides

$$\begin{pmatrix} Fib(n) \\ Fib(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} Fib(n-1) \\ Fib(n) \end{pmatrix}$$

Et donc

$$\begin{pmatrix} Fib(n) \\ Fib(n+1) \end{pmatrix} = M^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

On peut utiliser l'algorithme précédent aussi pour les puissances de matrices.

CHAPITRE 6

PROGRAMMATION DYNAMIQUE

Principe : mise en mémoire pour éviter les recalculs

Nous présentons d'abord quelques variantes simplifiées :

1. Mémoïsation
2. Récursivité ascendante
3. Programmation dynamique

Exemple : Nombres de Fibonacci

Les nombres de Fibonacci sont définis de la façon suivante :

$$Fib(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

Ou bien récursivement :

Fib(0) = 0

Fib(1) = 1

Fib(n+2) = Fib(n+1) + Fib(n)

Le temps d'exécution est :

$$\begin{aligned} T(0) &= \mathcal{O}(1) \\ T(1) &= \mathcal{O}(1) \\ T(n+2) &= T(n+1) + T(n) + \mathcal{O}(1) \end{aligned}$$

Il s'agit donc de la même récursion et donc d'un temps exponentiel :

$$T(n) = \mathcal{O}((\frac{1+\sqrt{5}}{2})^n)$$

6.1 Mémoïsation

6.1.1 Idée

Pour éviter les recalculs, on met les valeurs calculées dans un tableau.

6.1.2 Détail

En plus de la fonction de départ `f`, on introduit la fonction `fmémo` et on remplace tous les appels de `f` par des appels de `fmémo`. Celle-ci vérifie d'abord si le résultat ne se trouve pas dans son tableau ; sinon elle appelle `f` et met le résultat dans son tableau.

```
const undef = { une valeur constante jamais employée par ex −1 }

var FibTab : array [0..MAX] of integer; {initialisé à undef}
```

```
function Fibmémo (n : integer) : integer ;
{Pré : 0 ≤ n ≤ MAX }
begin
  if FibTab[n] = undef
  then FibTab[n] := Fib(n);
  Fibmémo := FibTab[n]
end;
```

Donc dans la fonction `Fib`, les appels récursifs se font à travers `Fibmémo`.

Pour calculer le temps d'une fonction mémoïsée, on multplie la taille de la partie utilisée du tableau (n pour `Fib`) par le temps d'un appel NON-récursif (constant pour `Fib`). Le temps de `Fib` après mémoïsation est donc linéaire.

Avantages de la mémoïsation

– Technique facile à programmer.
– La transformation peut être automatisée.
– L'ordre de grandeur du temps d'exécution décroît ou reste identique
– Seuls les sous-problèmes **utiles** sont calculés

Inconvénients

– Il faut limiter le nombre de valeurs possibles des arguments pour avoir un tableau de taille raisonnable.

6.2 Récursivité Ascendante

6.2.1 Idée

Pour éviter les test (... = undef) on s'arrange pour que la valeur se trouve déjà dans la tableau au moment où on en a besoin. Pour cela on calcule les appels en montant dans le graphe d'appels.

6.2.2 Avantages de la Récursivité Ascendante

– Petit gain de temps : élimination du test
– L'analyse du graphe d'appels permet souvent d'économiser de la mémoire.

6.2.3 Inconvénients

– La forme du programme est complètement changée.
– Attention aux calculs inutiles !

6.2.4 Sous-problèmes inutiles : Exemple

Logarithme entier : $\lfloor \log_k(n) \rfloor$ where $n \geq 1, n \in \mathbb{N}$.

Diviser pour régner donne :

$$ilog(n) = 0 \text{ si } 1 \leq n < k \tag{4}$$

$$ilog(n) = ilog(n\ div\ k) + 1 \text{ si } n \geq k \tag{5}$$

(6)

Le temps d'exécution est $\mathcal{O}(\log n)$.

6.2.4.1 Récursivité ascendante naïve

On aurait tendance à calculer $ilog$ pour tous les nombres jusque n :

```
for i := 1 to k−1 do
  ilogTab[i] := 0;
for i := k to n do
  ilogTab[i] := ilogTab[i div k] +1
```

Ce programme est plus lent $\mathcal{O}(n)$ et demande plus de mémoire $\mathcal{O}(n)$. En effet, il calcule et retient des sous-problèmes inutiles (qui n'interviennent pas dans le résultat final)

Les sous-problèmes *utiles* sont ceux qui ont un chemin d'appels depuis le problème de départ.

6.2.5 Économie de mémoire

1. La taille du tableau dépend du nombre de valeurs possibles des paramètres. Il faut donc réduire ceux-ci. Les paramètres qui ne changent pas (k pour $ilog$) peuvent être passés comme variable globale ou constante.
2. Il n'est souvent pas nécessaire de garder tout le tableau, car il se peut que certains éléments ne soient plus utilisés dans le futur. On peut les déterminer graphiquement :
   (a) On choisit un ordre de parcours
   (b) On trace une ligne séparant les éléments du tableau déjà calculés de ceux qui ne le sont pas encore
   (c) Les éléments calculés à conserver sont ceux qui sont le départ d'une flèche e qui traverse la ligne, autrement dit qui seront utilisés par la suite.

6.2.6 Exemple : Fibonacci

Graphe d'appel (ou de dépendances) : Fib(i) dépend de Fib(i-1) et Fib(i-2).

La récursivité ascendante appliquée à l'équation de Fibonacci donne :

```
var F: array[0..MAX] of natural;
F[0] := 0;
F[1] := 1;
for i:=2 to n do
    F[i] := F[i−2] + F[i−1];
Fib := F[n]
```

Temps et mémoire : $\mathcal{O}(n)$, comme la version mémoïsée.

6.2.6.1 Exemple : Fibonacci : Économie de mémoire

Le graphe d'appel montre qu'il ne faut garder que les deux derniers éléments en mémoire. Posons un nouveau tableau G avec l'invariant $G[i \textbf{mod} 2] = Fib(i)$.

```
var G: array[0..1] of natural;
G[0] := 0;
G[1] := 1;
for i:=2 to n do
    G[i mod 2] := G[0] + G[1];
Fib := G[n mod 2]
```

6.2.7 Exemple : Combinaisons

Le nombre de combinaisons (ensembles) de m éléments choisis dans un ensemble de taille n (avec $m \leq n$) est noté ici C_n^m.

Certains le notent $\begin{pmatrix} n \\ m \end{pmatrix}$ ou inversent m et n.

Pre : $n \geq m \geq 0$ Post : $C_n^m = \#\{s \subseteq \{1..n\} \mid \#s = m\}$

Diviser pour régner nous permet de trouver une définition récursive :

$C_n^m = 0$ Si $m > n$ ou $m < 0$ car il est impossible de prendre un nombre négatif d'objets ainsi que d'en prendre plus de n.

$C_n^0 = 1$ Car seul l'ensemble vide ne contient aucun élément.

$C_n^n = 1$ Car seul l'ensemble plein contient tous les objets.

$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$ Car, si on ajoute un élément supplémentaire ('n') dans un ensemble de $n-1$ éléments,

1. Toutes les anciennes combinaisons C_{n-1}^m restent valables (ce sont les cas où 'n' n'est pas pris).
2. Les combinaisons qui contiennent 'n' sont au nombre de C_{n-1}^{m-1} (car on prend $n-1$ anciens).

Le temps d'exécution est :

$$\mathcal{O}(1) \text{ pour les cas de base} \tag{7}$$

$$T(n) = 2T(n-1) + \mathcal{O}(1) \tag{8}$$

temps exponentiel en n.

On peut constater des recalculs dans l'arbre d'appels. Pour bien voir cela, on dessine le **graphe d'appels** où les mêmes appels sont fusionnés. Un noeud avec plusieurs pères est un appel recalculé plusieurs fois. On présente ce graphe dans le tableau de memoïsation.

Par récursivité ascendante, dans l'ordre des obliques :

```
function comb(m, n : integer) : integer;
   var C : array [0..MAX] of integer; ...
begin
  if m > n div 2 then  m := n-m;
  for j := 0 to m do C[j] :=1;
  for i := 0 to n-m-1  do
      for j := 1 to m do
          C[j] := C[j] + C[j-1]
  comb := C[m];
end;
```

Remarques

1. Le test $if\ldots$ a pour but de maintenir m plus petit, et donc d'économiser du temps et de la place mémoire.
2. L'invariant de la boucle for imbriquée (donc celle en j) est

$$k < j \quad \Rightarrow \quad C[k] = C^k_{i+k+1} \tag{9}$$

$$k \geq j \quad \Rightarrow \quad C[k] = C^k_{i+k} \tag{10}$$

3. Le temps d'exécution est $\mathcal{O}(n * m) \leq \mathcal{O}(n^2)$
4. La place mémoire est $\mathcal{O}(m)$ (linéaire, au lieu de $\mathcal{O}(n^2)$ pour une mémoïsation simple).

Ceci est juste une illustration historique de la programmation dynamique : on peut faire plus rapide (linéaire) et plus économe en mémoire (constante) en se ramenant à la factorielle.

6.3 Programmation dynamique

La programmation dynamique combine “diviser pour régner”, puis la récursivité ascendante avec mise en mémoire.

On l’applique surtout sur des problèmes d’optimisation : trouver une solution faisable de coût minimal (ou de gain maximal).

1. Trouver la récursion par « diviser pour régner ».
   (a) Dans le cas d’un problème d’optimisation, on l’appelle propriété de sous-solution optimale : montrer que le problème de départ peut se calculer à partir des solutions optimales de sous-problèmes plus petits.
   (b) Pour simplifier on ne renvoie que le coût optimal (au point 4 on reconstruira la solution complète.)
   (c) limiter les paramètres pour réduire la taille du tableau.

2. Récursivité ascendante : de la définition récursive, on déduit le graphe des dépendances.
   (a) Si plusieurs chemins partent du même appel, il y a des recalculs donc on met en mémoire. Sinon, la récursivité simple suffit.
   (b) On choisit un ordre de calcul compatible avec les dépendances.
   (c) On minimise l'emploi de mémoire en supprimant la mémoire qui ne sera plus utilisée, autrement dit on ne garde que les départs d'une flèche qui franchit la frontière de calcul.
   (d) On reconstruit la solution : Chaque choix est mémorisé dans un tableau ; à la fin, on reconstruit en arrière une solution optimale grâce à ces choix.

6.3.1 Multiplication d'une suite de matrices

Rappel

Les matrices se multiplient "ligne par colonne"

SI $A = (a_{ij})_{i=1..n,j=1..m}$ est de taille $n \times m$

$B = (b_{jk})_{j=1..m,k=1..p}$ est de taille $m \times p$

ALORS $A * B = (\sum_{j=1}^{m} a_{ij} * b_{jk})_{i=1..n,k=1..p}$ est de taille $n \times p$

et est calculé en $n.m.p$ multiplications.

La multiplication matricielle est associative : on peut choisir de mettre les parenthèses comme on veut, mais elle n'est pas commutative.

On omet souvent d'écrie la multiplication : $AB = A * B$.

Spécification

On donne un tableau contenant les dimensions p_i d'une suite de n matrices

p: **array**[0..n] **of** `natural`; $M_i : p_{i-1} \times p_i$

On demande un parenthésage, càd une façon de mettre les parenthèses dans

$$M_1 * M_2 * \ldots * M_n$$

qui minimise le nombre total de multiplications effectuées.

Concepts dans cette spécification

Les parenthésages de $M_i * M_2 * \ldots * M_j$ sont

$$\begin{aligned} PAR(i,j) &= \{"("P_{i,k}")" * ("P_{k+1,j}")" \mid k < j, k \geq i, P_{i,k} \in PAR(i,k), P_{k+1,j} \in \\ PAR(i,i) &= \{M_i\} \end{aligned}$$

Le nombre de multiplications d'un parenthésage est

$$\begin{aligned} nmul((P_{i,k}) * (P_{k+1,j})) &= nmul(P_{i,k}) + nmul(P_{k+1,j}) + p_{i-1}.p_k.p_n \\ nmul(M_i) &= 0 \end{aligned}$$

Exemple

$$p = [10, 1, 10, 10, 1, 10]$$

$$M_1 : 10 \times 1,\ M_2 : 1 \times 10,\ M_3 : 10 \times 10,\ M_4 : 10 \times 1,\ M_5 : 1 \times 10$$

$(((M_1M_2)M_3)M_4)M_5$			$M_1(((M_2M_3)M_4)M_5)$		
$rs.$	$\#mult.$	$dim.rs.$	$rs.$	$\#mult.$	$dim.rs.$
$M_1M_2:$	100	10×10	$M_2M_3:$	100	1×10
$\ldots M_3:$	1000	10×10	$\ldots M_4:$	10	1×1
$\ldots M_4:$	100	10×1	$\ldots M_5:$	10	1×10
$\ldots M_5:$	100	10×10	$M_1 \ldots:$	100	10×10
$Total:$	1300		$Total:$	220	

DIviser pour règner

Soit $m(i,j)$ le coût optimal, càd le nombre minimum de multiplications nécessaires pour calculer $M_{i,j} = M_i * M_{i+1} \ldots * M_j$

$m(i,j) = \min_{p \in PAR(i,j)} nmul(p)$

Notre objectif de départ s'écrit donc $m(1,n)$.

La mesure bien fondée est le nombre de multiplication matricielles $j - i$.

Cas de base : $m(i,i) = 0$.

Pour $j > i$ on a la *propriété de sous-solution optimale* :

$$m(i,j) = \min_{i \leq k < j} \{m(i,k) + m(k+1,j) + p_{i-1}.p_k.p_j\}$$

Graphe des appels

p = [10, 1, 10, 10, 1, 10]

i	1	2	3	4	5	j
1	0	100	200	120	220	
2		0	100	110	120	
3			0	100	200	
4				0	100	
5					0	

Chaque appel $m(i, j)$ dépend de tous ceux à gauche $m(i, k)$ et de tous ceux en-dessous $m(k, j)$. Il y a bien des recalculs. Il faut donc faire les calculs par diagonale (i.e. par valeur de variant $j - i$ croissante). Les dépendances qui traversent la frontière de calcul nous amènent à tout conserver.

6.3.1.1 Reconstruction de la solution

On retient le *choix* du meilleur k dans un tableau c lors du calcul du minimum.

```
procedure imprimeParenthésage(c: tabchoix; i,j : integer);
var k: integer;
begin
    k := c[i,j];
    write("(");
    imprimeParenthésage(c, i,k);
    write(") * (");
    imprimeParenthésage(c, k+1,i);
    write(")");
end
```

6.3.1.2 Programme

```
for i := 1 to n do m[i,i] := 0;
for v := 1 to n-1 do
    for i := 1 to n-v do begin
        j := i+v; b := maxint;
        for k := i to j-1 do begin
            t := m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j];
            if t < b then begin b := t; c[i,j] := k end
        end;
        m[i,j] := b;
    end;
end;
result := m[1,n]
```

Temps : $\mathcal{O}(n^3)$
Mémoire : $\mathcal{O}(n^2)$

L'énumération de tous les parenthésages (« générer et tester ») demanderait un temps exponentiel, car le nombre de parenthésages se calcule comme :

$$\begin{aligned} nbPAR(v) &= \sum_{v_1+v_2+1=v, v_1, v_2 \geq 0} nbPAR(v_1) * nbPAR(v_2) \\ nbPAR(0) &= 1 \end{aligned}$$

$nbPAR(n)$ est le nombre de Catalan C_n.

Il y a un algorithme en $\mathcal{O}(n \log n)$:
Hu, T C. ; M T. Shing (1984). "Computation of matrix chain products." SIAM Journal on Computing 13 (2) : 228–251.

6.3.2 Exemple : sac à dos discret

Un voleur pénètre par effraction dans un magasin. Il vous demande comment remplir son sac à dos pour maximiser la valeur du contenu ?

Données

- Charge utile (ou capacité) du sac : $C \in \mathbb{N}$ (nombre naturel)
- Il y a n articles dans le magasin. Pour chaque article du magasin $i \in 1..n$ on a :
  - sa valeur : $v_i \in \mathbb{N}$ (nombre naturel)
  - son poids : $t_i \in \mathbb{N}^+$ (nombre naturel positif)

Résultat

Donner un tableau $q_i \in \mathbb{N}$ (quantités emportées) qui maximise la valeur V du contenu :

$$V = \sum_{i=1}^{n} q_i * v_i$$

sous la contrainte que le poids total ne dépasser pas la capacité :

$$\sum_{i=1}^{n} q_i * t_i \leq C$$

Formalisation récursive

Il y a 2 façons de créer des problèmes plus petits :

1. un plus petit magasin : diminuer n
2. un plus petit sac : diminuer C

Le On pose $g(k, c)$ = gain maximum pour l'ensemble d'articles $\{1 \ldots k\}$ et une capacité $c \in 0..C$

Cas de base :

– Si le sac n'a pas de capacité, on ne peut rien y mettre : $g(k, 0) = 0$
– Si le magasin est vide, on ne peut rien emporter : $g(0, c) = 0$

6.3.3 D1 : Diviser en un élément / le reste

$$
\begin{aligned}
g(k,c) = \quad & \text{SI } c < t_k \text{ ALORS } g(k-1,c) \\
& \text{SINON } \max\{g(k-1,c), v_k + g(k,c-t_k)\} \text{ où } k > 0 \\
g(0,c) = \quad & 0 \\
g(k,0) = \quad & 0
\end{aligned}
$$

c	0	1	2	3	$\ldots$	$C-1$	C
n	0	$g(n,1)$	$g(n,2)$	$g(n,3)$	$\ldots$	$g(n,C-1)$	$g(n,C)$
$n-1$	0	$g(n-1,1)$	$g(n-1,2)$	$g(n-1,3)$	$\ldots$	$g(n-1,C-1)$	$g(n-1,$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		
3	0	$g(3,1)$	$g(3,2)$	$g(3,3)$	$\ldots$		$g(3,C)$
2	0	$g(2,1)$	$g(2,2)$	$g(2,3)$	$\ldots$		$g(2,C)$
1	0	$g(1,1)$	$g(1,2)$	$g(1,3)$	$\ldots$		$g(1,C)$
0	0	0	0	0	0		

6.3.3.1 Conséquences

L'implémentation récursive implique des recalculs Ordre ascendant choisi :

– ligne par ligne, de gauche à droite

– Une ligne suffit en mémoire :

```
var gain : array[0..C] of integer;
```

6.3.3.2 Invariants

– de *for k := 1 to n do ...* (boucle extérieure $\rightarrow$ pour chaque ligne)

$$0 \leq k \leq n$$

$$\forall j : 0 \leq j \leq C : gain[j] = g(k, j)$$

– de *for j := 1 to C do ...* (boucle intérieure $\rightarrow$ pour chaque élément de la ligne)

$$0 < k \leq n$$

$$0 \leq j \leq C$$

partie traitée $\forall i : 0 \leq i < j \Rightarrow gain[i] = g(k, i)$

partie non traitée $\forall i : j \leq i \leq C : gain[i] = g(k-1, i)$

6.3.3.3 Programme

```
for j := 0 to C do gain[j] := 0;
for k := 1 to n do
    for j := 1 to C do
       if j >= t[k]
      then gain[j] := max(gain[j], v[k]+ gain[j-t[k]])
```

L'application des techniques de programmation dynamique dans ce cas précis donnent donc

Espace mémoire : tableau de taille C $\rightarrow \mathcal{O}(C)$.

Temps d'exécution : n passages $\rightarrow \mathcal{O}(nC)$.

Ce qui est coûteux dans les cas où C est grand.

6.3.4 Amélioration gloutonne

Si il y a un article i et un article $j (i \neq j)$ tels que

$$\exists q : q * t_i \leq t_j \text{ et } q * v_i \geq v_j$$

Alors j ne sera jamais employé[a]. En généralisant cette constatation :

Si $\exists q_i, q_j$: $q_i * t_i \leq q_j * t_j$ [plus léger]

$q_i * v_i \geq q_j * v_j$ [plus cher]

Alors j sera employé moins de q_j fois, car sinon on le remplace par i.

a. Car j est moins cher et plus lourd que i

L'idée est donc de calculer le "rapport qualité/prix" v_i/t_i. Soit m l'article de meilleur rapport ; tous les autres ont une borne q_j telle que

$$q_j \leq t_m \quad \text{car } t_j * t_m \leq t_m * t_j$$
$$\text{et } t_j * v_m \geq t_m * v_j$$

Par application des relations de la généralisation et car m est le meilleur rapport.
De même,

$$q_j \leq v_m$$

Soit $C_m = \sum_{j \neq m} q_j * t_j$; on ne calcule la table “bouche-trou” que jusque C_m.

Si $C > C_m$:
- Prendre $\lceil \frac{C - C_m}{t_m} \rceil$ articles de type m.
- Regarder dans la table pour le reste.

6.3.4.1 Correct ?

1. Si $C > C_m$, la solution contient un article m (puisque les autres sont bornés).
2. $g(S, C) = g(S, C - t_m) + v_m$ si $C > C_m$.

CHAPITRE 7

ALGORITHMES GLOUTONS

Trouver localement un choix qu'on peut toujours faire dans un optimum

Exemples

Sac à dos continu borné* Prendre d'abord les articles de meilleur rapport valeur/poids.

Pleins* Faire le plein le plus tard possible

Salle de spectacle * Prendre l'activité qui se termine le plus tôt.

Codes de Huffman * Fusionner les sous-arbres de moindre fréquences.

Voyageur de commerce Commencer par la ville la plus proche.

Correct ?

Pour les problèmes marqués d'une '*' la solution est optimale (et sinon il donne une solution faisable, mais pas optimale).

Preuve

Le schéma d'une preuve gloutonne a deux étapes ; on prouve que :

1. Il y a un optimum qui contient le premier choix fait. Pour cela, on suppose un optimum, et on montre que "notre" solution a la même valeur d'objectif
2. L'optimum pour le reste peut se trouver récursivement
3. En le combinant avec le premier choix, on trouve l'optimum.

C'est une preuve de type C1 : trouver un élément du résultat, puis le reste.
Elle donnera souvent une récursion terminale équivalente à une boucle.

7.1 Sac à dos

7.1.1 Continu

Dans ce cas, les q_i peuvent être fractionnaires.

Ici on a toujours intérêt à prendre l'article m de meilleur rapport v_m/t_m :

$\Longleftrightarrow q_m = C_0/t_m$

Temps linéaire (pour trouver l'article m), mémoire constante.

7.1.2 Sac à dos continu borné

On impose une limite sur la quantité présente dans le magasin : $q_i \leq Q_i$

$\Rightarrow$ Prendre les articles selon les rapports v_i/t_i décroissants jusqu'à remplir le sac ou vider le magasin.

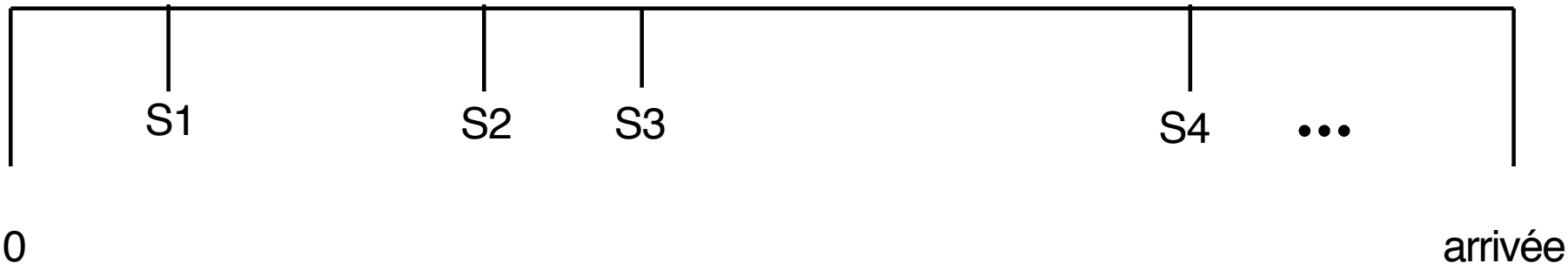
Temps $\mathcal{O}(n \log n)$ (pour trier les articles).

7.2 Exemple : Les pleins d'essence

Donald a planifié son itinéraire de vacances, le long duquel il a relevé le kilométrage des stations d'essence. Il veut s'arrêter le moins possible, sans tomber en panne sèche.

Données

- Un réservoir de capacité $C_M \geq 0$ (en kilomètres)
- La distance à parcourir $L \geq 0$
- La position des n stations le long de la route : $(S_i)_{i=1..n}$ où $0 \leq S_i < S_{i+1} < \ldots < L$



- Le contenu du réservoir au départ : C_0, où $0 \leq C_0 \leq C_M$.

Résultat

Un tableau q contenant les quantités achetées q_i à chaque station i, avec $0 \leq q_i \leq C_M$, ou signaler qu'il n'y a pas de solution faisable. La fonction $C(l)$ donne le contenu du réservoir en fonction du kilométrage : $C(l) = C_0 + \sum_{S_i<l} q_i - l$. Si l est la position d'une station, il s'agit du contenu à l'entrée de la station. Le contenu du réservoir en sortant de la station k est donc : $C_k = C(S_k) + q_k$. Les arrêts A sont alors les stations où on achète de l'essence : $A = \{i \in 1..n | q_i > 0\}$.
Le problème est de minimiser le nombre d'arrêts $\left|A\right|$ sans tomber en panne avant de rallier l'arrivée :

$$\forall l, 0 \leq l < L : C(l) \in [0..C_M]$$

Nous avons 2 caractéristiques gloutonnes :

7.2.1 Si on s'arrête, autant faire le plein :

Soit q un optimum. q' modifie q en faisant le plein chaque fois que q s'arrête : $q'_i = C_M - C'(S_i)$ si $q_i > 0$. Alors q' est aussi optimale.

Preuve : $A' = A$ donc la valeur de l'objectif est la même ; q' reste faisable (on ne tombe pas en panne) car $C_M \geq C'(l) \geq C(l) \geq 0$.

7.2.2 Si on a de quoi atteindre la prochaine station, pas d'arrêt :

Soit q un optimum. Si $C(l) \geq (S_{k+1} - S_k)$, alors soit q' une solution où $q'_k = 0$ (on ne s'arrête pas en k) et $q'_{k+1} = q_k + q_{k+1}$ (on achète à la station suivante). La fonction $C'(l)$ est identique à $C(l)$, sauf entre $S_k et S_{k+1}$, où $C'(l) = C(l) - q_k$, mais elle satisfait toujours la contrainte.

Programme

Etant donné le choix de la première station, on résoud récursivement avec une station en moins avec le réservoir initial soit plein si on s'arrête, ou diminué du trajet parcouru sinon. Soit C le tableau du contenu du réservoir en sortant de la station précédente. On pose $S_0 = 0$.

$C := C_0$;
for i := 1 **to** n **do**
  if $C < (S_i - S_{i-1})$ *(* il faut pouvoir atteindre la station suivante *
  then panneDessence
  else if $(S_{i+1} - S_{i-1}) > C$ *(* peut–on atteindre la station d'après? *)*
    then begin *(* si non, on fait le plein *)*
      $q_i = C_M - (C - (S_i - S_{i-1}))$;
      $C := C_M$
    end
    else begin *(* si oui, on passe *)*
      $q_i := 0$;
      $C := C - (S_i - S_{i-1})$
    end

Temps : $\mathcal{O}(n)$.

Mémoire : $\mathcal{O}(1)$ si on ne compte pas les données S_i ni les résultats $q_i(\mathcal{O}(n))$.

7.2.3 Variante avec prix

Données On ajoute le prix à chaque station $p_i > 0$.

Résultat Le coût à minimiser est : $\sum_{i \in 1..n} p_i * q_i$

Note : cet objectif encourage à arriver avec un réservoir vide.

Choix glouton : remplir le plus possible à la station m la moins chère. On arrive à vide (sauf si C_0 permet d'atteindre m) et on fait le plein (sauf si on peut atteindre la destination).

On fait deux appels récursifs, pour résoudre le trajet avant m et le trajet après m.

Temps : retrouver la station la moins chère est linéaire, ce qui mène a un coût total quadratique.

Une solution : un arbre rouge-noir trié sur le kilométrage des stations et augmenté par le prix minimum des sous-arbres permet de retrouver la station de prix minimum entre S_i et S_j en temps $\mathcal{O}(n \log n)$.

Autre choix glouton : On prend parmi toutes les stations accessibles, la moins chère s_m. De là, on calcule de nouveau la moins chère s_n. Si le prix est meilleur en s_n, on prend juste de quoi l'atteindre, sinon on fait le plein.

7.3 Exemple : Salle de spectacle

Données

Une liste d'activités candidates avec pour chacune :

s_i : temps de début

f_i : temps de fin (avec $s_i \leq f_i$)

Résultat

Un ensemble $A \subset \{1 \ldots n\}$ de taille maximale d'activités compatibles [a]

$$\iff \forall i, j \in A : s_i \geq f_j \; ou \; s_j \geq f_i$$

a. i.e. chronologiquement disjointes

Idée de preuve

1. Une solution optimale contient l'activité qui se termine le plus tôt (notée k).

   Soit A une solution optimale et soit m sa première activité.

   $$B = A \setminus \{m\} \cup \{k\} \text{ est faisable et optimale.}$$

2. Après avoir choisi k, on résoud pour les activités compatibles avec k.

Algorithme

– Trier les activités par temps de fin ($\mathcal{O}(n \log n)$).

– $A := \emptyset$; $f := -\infty$

Pour chaque activité i prise dans cet ordre ($\mathcal{O}(n)$)

if $s_i \geq f$

then $A := A \bigcup \{i\}$ et $f := f_i$

Temps total : $\mathcal{O}(n \log n)$

7.4 Codes de Huffman

7.4.1 Codages des caractères

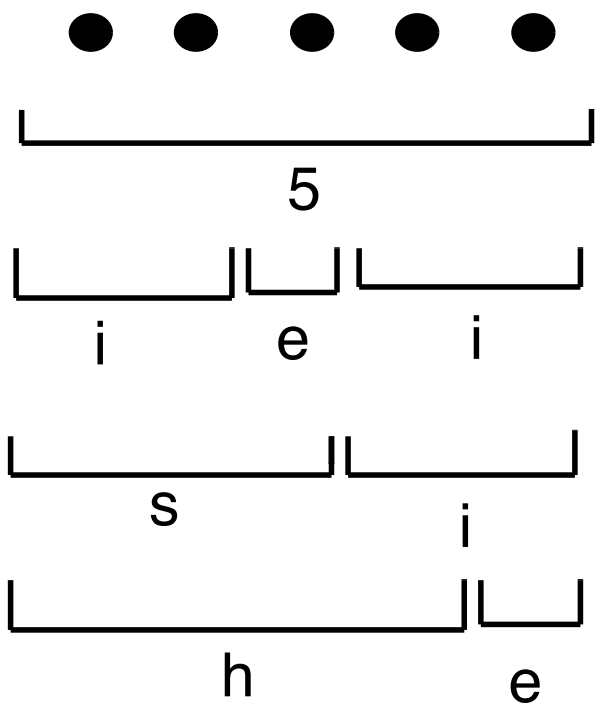
7.4.1.1 Fixe

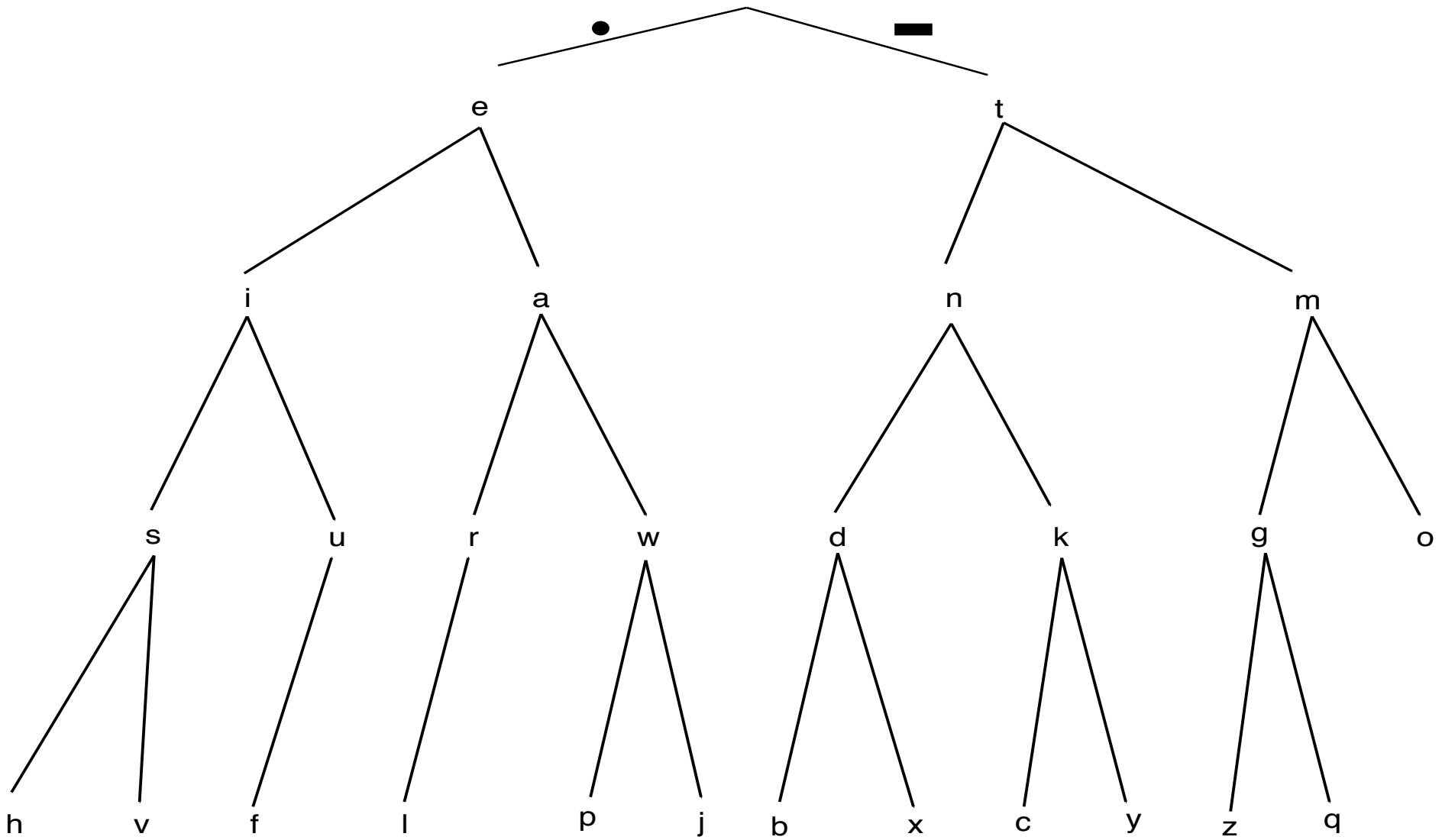
p.ex. ISO-Latin-1 fixe 8 bits pour chaque caractère,

7.4.1.2 Morse

les caractères plus fréquents (par ex. ’E’) ont un code plus court.

Le problème du morse est de séparer deux caractères :


5
i
e
i
s
i
h
e


•
–
e
t
i
a
n
m
s
u
r
w
d
k
g
o
h
v
f
l
p
j
b
x
c
y
z
q

7.4.2 Calcul du code optimal

Données

– Un alphabet, càd un ensemble de caractères C.

– Pour chacun, une fréquence $f[c]$

Résultat Un code (une suite de bits) $e[c]$ pour chaque caractère tel que aucun code n'est préfixe d'un autre et $\sum_{c \in C} d(c) * f[c]$ (= longueur du texte) est minimum où $d(c)$ est la longueur de $e[c]$.

Exemple

$$C = \{a, b, c, d, e, f\}$$

$$f = 45, 13, 12, 16, 9, 5.$$

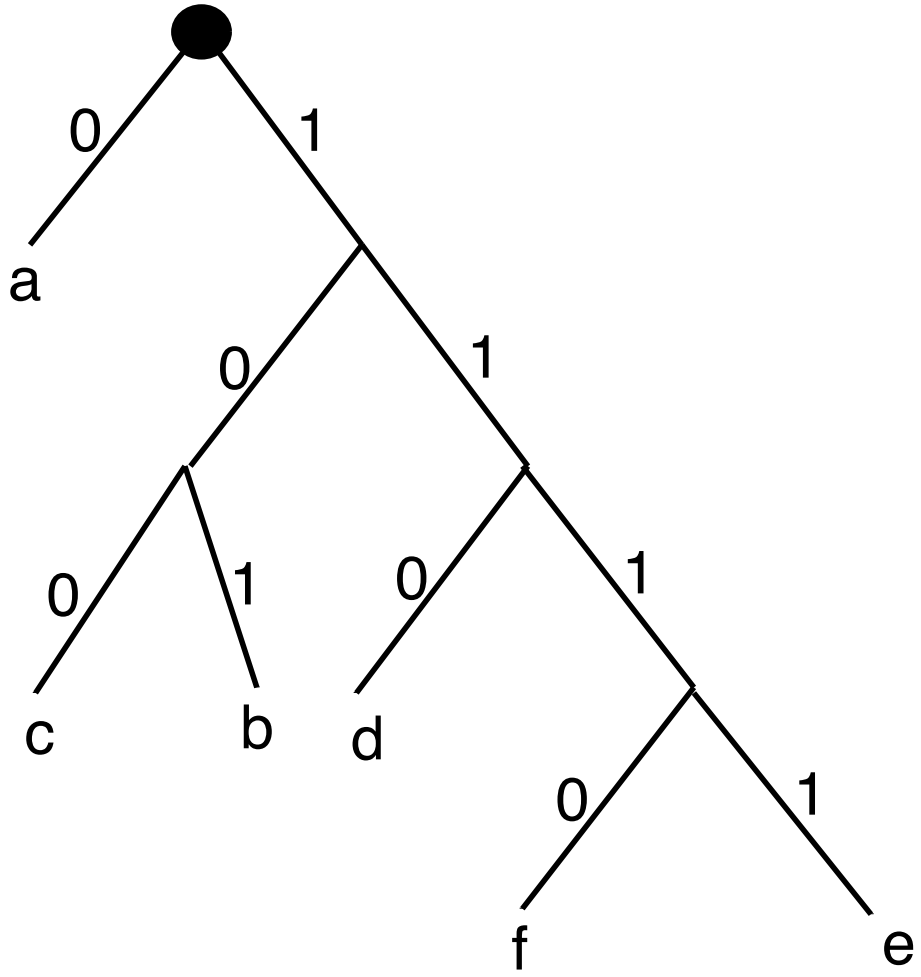
codage ISO-Latin-1 8 * 100 = 800 bits.

codage fixe sur 3 bits 3 * 100 = 300 bits. [a]

codage optimal 224 bits

a	b	c	d	e	f
0	101	100	110	1111	1110

a. car 3 = $\lceil \log_2 6 \rceil$


0
1
a
0
1
0
1
0
1
c
b
d
0
1
f
e

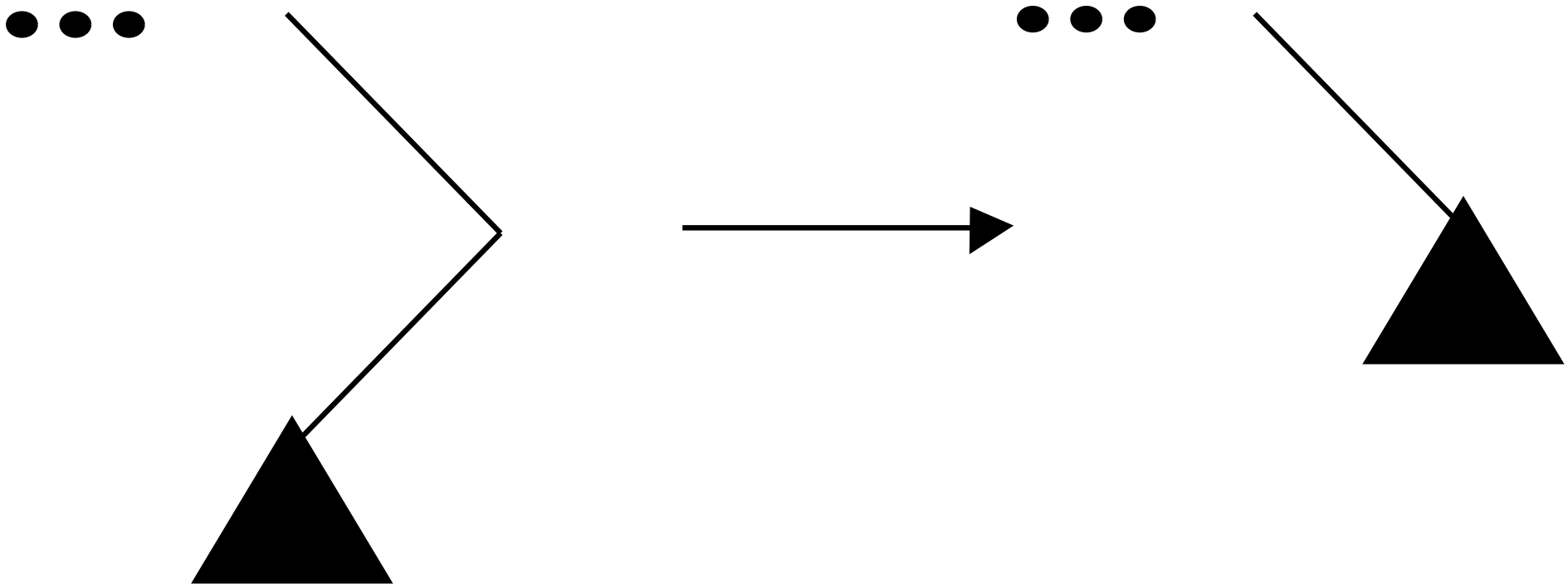
Lemmes

7.4.2.1 Sans préfixe

Un code est sans préfixe ssi un caractère n'apparaît que sur une feuille de l'arbre.

7.4.2.2

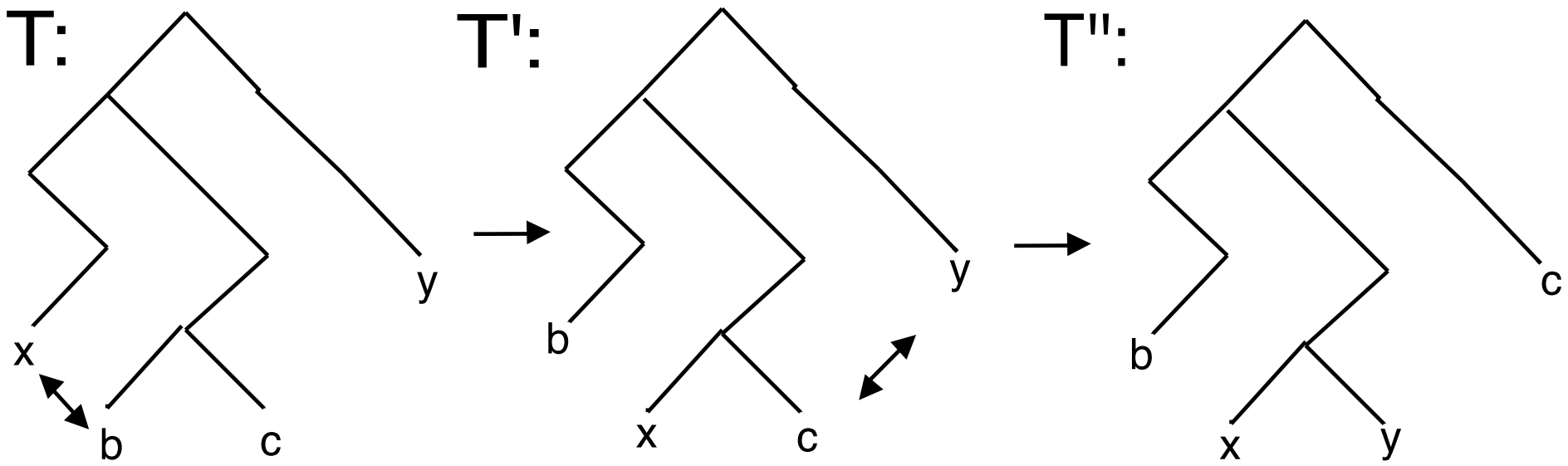
Dans un code sans préfixe optimal un noeud a 0 ou 2 fils. En effet, dans le cas contraire, on peut "raccourcir la branche".



7.4.2.3 Moins fréquents

Si x, y sont les deux caractères les moins fréquents alors il existe une solution optimale où x et y ont deux codes les plus longs et diffèrent seulement par le dernier bit.

Preuve :
Soit b, c deux codes les plus longs qui ne diffèrent que par le dernier bit.



coût(T) - coût(T') $= \sum_c f[c] * d(c) - \sum_c f[c] * d'(c)$

$= f[x] * d(x) + f[b] * d(b) - f[x] * d'(x) - f[b] * d'(b)$

Or $d'(x) = d(b)$ et $d'(b) = d(x)$ $= f[x] * d(x) + f[b] * d(b) - f[x] *$

$= (f[b] - f[x]) * (d(b) - d(x))$

$= (\geq 0) * (\geq 0)$ car x est le moins fréquent et b a le code le plus long

7.4.2.4 Récursion

Après avoir regroupé les deux caractères les moins fréquents, peut-on se ramener à un problème similaire ?

Soit T une solution, x, y deux caractères frères de T.
Alors T est optimal ssi T$\backslash\{x, y\} \bigcup \{z\}$ est optimal pour le problème avec $C' = C\backslash\{x, y\} \bigcup \{z\}$ où z est un "nouveau" caractère (un arbre formé des feuilles x, y, de fréquence $f[x] + f[y]$.

Preuve :

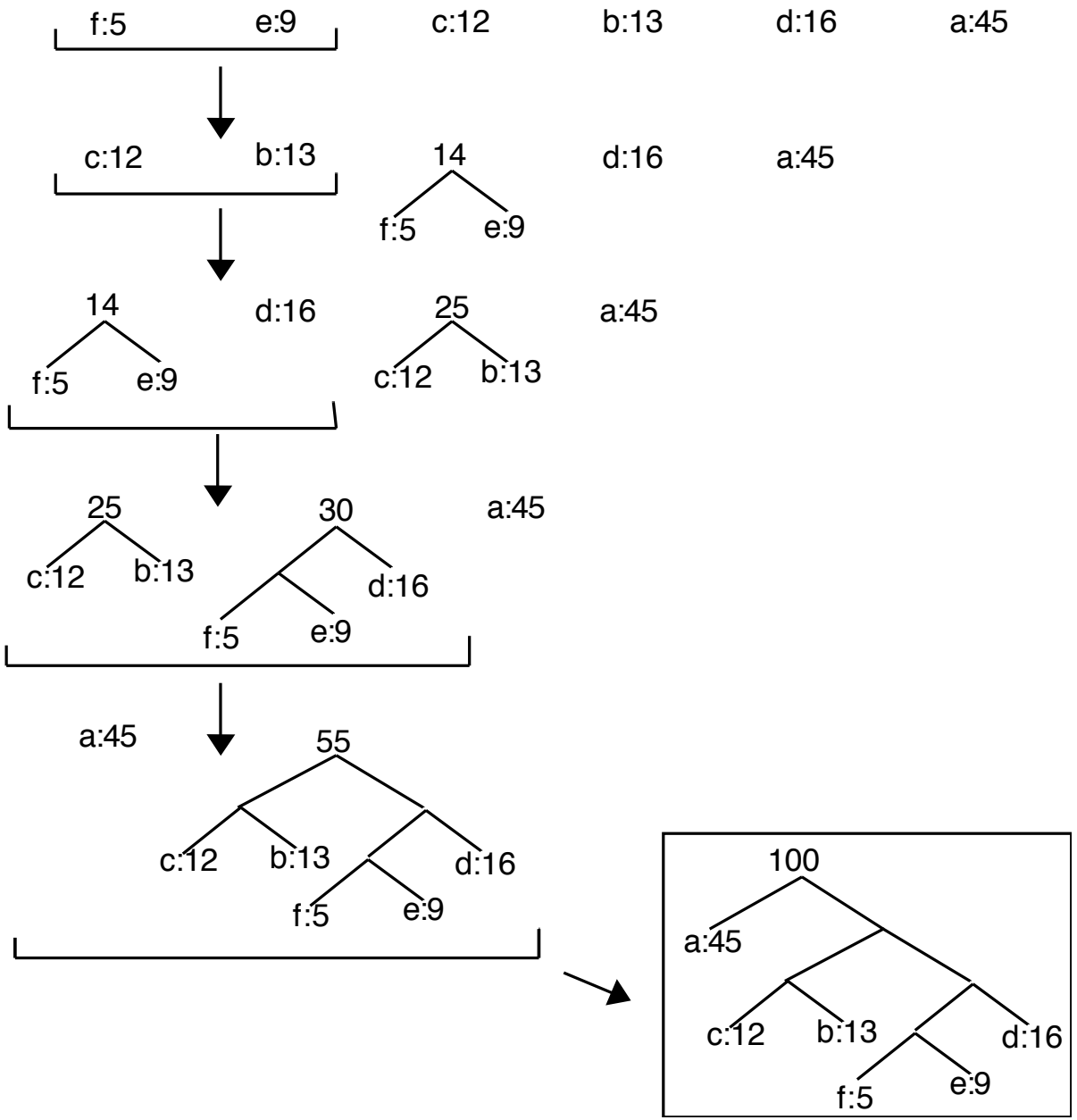
$$\begin{aligned}
\text{coût}(T) &= \sum_{c \in C \setminus \{x,y\}} f[c] * d(c) + d(x) * (f[x] + f[y]) \text{ car } d(x) = d(y) \\
&= (\sum_{c \in C'} f[c] * d(c)) + f[x] + f[y] \text{ car } f[z] = f[x] + f[y] \text{ et } d(z) + 1 = d(x) \\
&= \text{coût}(T') + (f[x] + f[y])
\end{aligned}$$

La différence ne dépend que des données : minimiser T revient à minimiser T'.

7.4.2.5 Algorithme

Tant qu'il reste au moins deux caractères :

1. Retirer les deux caractères les moins fréquents : x et y.
2. Insérer l'arbre ayant comme fils x et y et comme fréquence $f[x] + f[y]$


f:5
e:9
c:12
b:13
d:16
a:45
c:12
b:13
14
d:16
a:45
f:5
e:9
14
d:16
25
a:45
f:5
e:9
c:12
b:13
25
30
a:45
c:12
b:13
d:16
f:5
e:9
a:45
55
c:12
b:13
d:16
f:5
e:9
100
a:45
c:12
b:13
d:16
f:5
e:9

7.4.2.6 Implémentation

Comment trouver efficacement les deux caractères les moins fréquents ?

Une solution est le TA file de priorité, en particulier l'implémentation en tas qui permet une complexité en $\mathcal{O}(\log(n))$ pour l'ajout et le retrait et en temps constant pour la recherche du minimum.

La complexité d'une telle implémentation sera donc de $\mathcal{O}(nlog(n))$, car on passe n fois dans la boucle.

7.5 Exposants

Ecrire le produit de valeur maximale en utilisant des bases b et des exposants e donnés.

Exemple : si les bases sont 3,5,7 et les exposants 1,2,4, on peut par exemple écrire le produit $7^1 * 5^2 * 3^4 = 14175$.

Données Une constante entière $N > 0$;

Un tableau d'entiers positifs b (bases) indexé de $1..N$;

Un tableau d'entiers positifs e (exposants) indexé de $1..N$.

Résultat Une permutation p de $1..N$ qui maximise

$$\prod_{i=1}^{N} b[p[i]]^{e[i]}$$

Propriété En étendant les exposants, le gain s'écrit :
$b[p[1]] * b[p[1]] * .... * b[p[N]] * b[p[N]]$. Le nombre total de facteurs est fixe : $\sum_{i=1}^{N} e[i]$. Si $b[p[i]] > b[p[j]]$ et $e[i] < e[j]$, en échangeant i et j dans p, le gain augmente car on remplace $e[j] - e[i]$ facteurs $b[p[j]]$ par $b[p[i]]$ qui est plus grand. Donc une permutation optimale respecte l'ordre croissant des b et des e. On la trouve en triant b et e.

Temps : $\mathcal{O}(n \log n)$

7.6 Le voyageur de commerce

L'idée d'algorithmes gloutons donne parfois des heuristiques (i.e. des solutions bonnes mais non optimales)

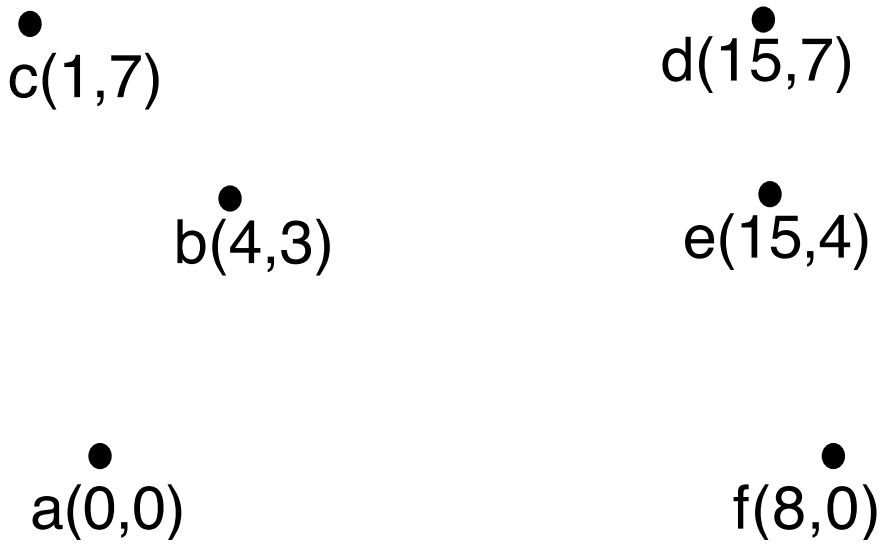
Problème Un représentant de commerce doit visiter n clients puis rentrer chez lui, par le plus court chemin possible.

Données Un ensemble de "villes" $V = \{1, \ldots n\}$ et leurs distances : $d : V \times V \to R^+$.

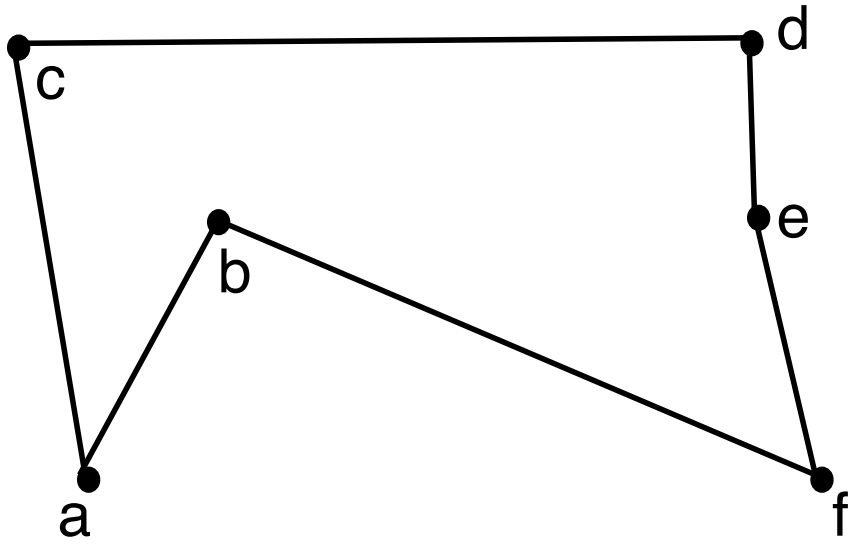
Résultat Une permutation p des villes telle que

$\sum_{i=0}^{n-1} d(p_i, p_{i+1}) + d(p_n, p_0)$ est minimal.

Exemple Numérique Considérons les villes suivantes, chacune munie de sa position, le tout sous la distance euclidienne.



Le résultat sera : $[a, b, f, e, d, c]$ pour un coût (optimal) de 48.39.


d
c
e
b
a
f

Heuristiques gloutonnes

7.6.0.7 Plus proche

Partant de son domicile, le représentant va toujours à la ville la plus proche non encore visitée.

Exemple :

Dans cet exemple, le coût d'une telle technique est de 50 > coût optimal, car le dernier trajet est souvent long. [a]. Le temps d'exécution est de l'ordre de $\mathcal{O}(n^2)$

a. Mais si on partait de b alors on obtiendrait la solution optimale

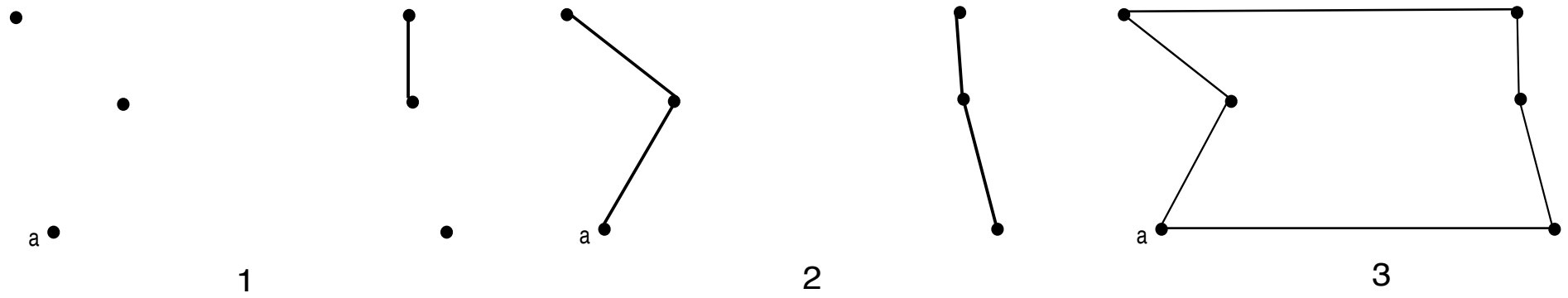
7.6.0.8 Plus proche avant/arrière

On agrandit aussi le trajet “en arrière” si c’est moins coûteux.

Sur l’exemple : le même résultat.

7.6.0.9 Par arcs

On prend les arcs dans l'ordre de longueur croissante , a condition que cela ne crée pas de cycle ni de carrefour (degré > 2).



Si les arcs sont dans un tas, on trouve une complexité de $\mathcal{O}(n^2 \log n)$ car il y n^2 arrêtes.

CHAPITRE 8

GÉNÉRER ET TESTER

Idée : essayer toutes les solutions.

Pour un problème d'optimisation, on garde la meilleure.

8.1 Principe

On divise la spécification du problème en deux parties :

1. la structure d'une *solution*, qu'on va générer
   (a) On la découpe en *choix*, pour lesquelles on a des *alternatives*
   (b) Un ensemble d'alternatives est une solution partielle ou un *état*
2. Les contraintes, qu'on va tester.

Exemple : N reines

On veut trouver toutes les façons de placer N reines sur un échiquier $N \times N$ sans qu'elles ne puissent se prendre, càd. qu'elles ne peuvent être sur la même ligne, colonne, ou diagonale.

Structure d'une solution :

1. tableau des positions des reines : donne N^{2N} solutions
2. chaque reine étant identifiée par sa colonne, on choisit sa ligne : donne N^N solutions ; on ne doit plus vérifier la contrainte sur les colonnes

8.2 Génération

1. Un programme récursif permettra de générer toutes les solutions
2. chaque niveau de récursion reçoit une solution partielle (état) et essaie toutes les alternatives d'un nouveau choix : La pile de récursion contient l'information utile pour continuer la génération. Son exécution peut être représentée par un arbre d'exploration. La récursion classique l'explore en profondeur (DFS).
3. Lorsque la solution courante est complète, il teste les contraintes
4. Pour un problème d'optimisation, on évalue la valeur de l'objectif et on mémorise la solution courante si c'est la meilleure jusqu'ici.

Exemple : N reines

Structure d'une solution partielle : pour les k premières colonnes, on donne dans quelle ligne se trouve la reine de cette colonne. On la représente par une liste en ordre inverse. L'appel initial se fait avec la liste vide (nil).

Choix suivant : la ligne de la reine dans la colonne $k+1$. Les alternatives sont les nombres de 1 à N.

```
procedure sols(l: liste)
begin
   if length(l) < N then for i := 1 to N do sols(cons(i,l))
   else if test(l) then printQueen(l)
end
```

8.3 Améliorations

Elagage (pruning) : On a intérêt à évaluer les contraintes dès que possible sur une solution partielle, soit pour éliminer toutes les solutions descendantes.

Propagation : Trouver des conséquences plus simples des contraintes et des choix déjà faits permet de les évaluer plus tôt.

Domaine : En particulier, on peut retenir les alternatives encore possibles pour un choix. S'il n'y en a plus qu'une, on la prend.

Pour les 4 reines, supposons qu'on choisisse pour chaque colonne de gauche à droite.

Elagage : La première solution générée est 1,1,1,1, ensuite 1,1,1,2, etc. mais on aurait pu backtracker dès qu'on a construit la solution partielle 1,1 puisque les deux premières reines sont sur la même ligne.

Propagation :

Domaine : On retient pour chaque colonnes les cases qui sont encore possibles.

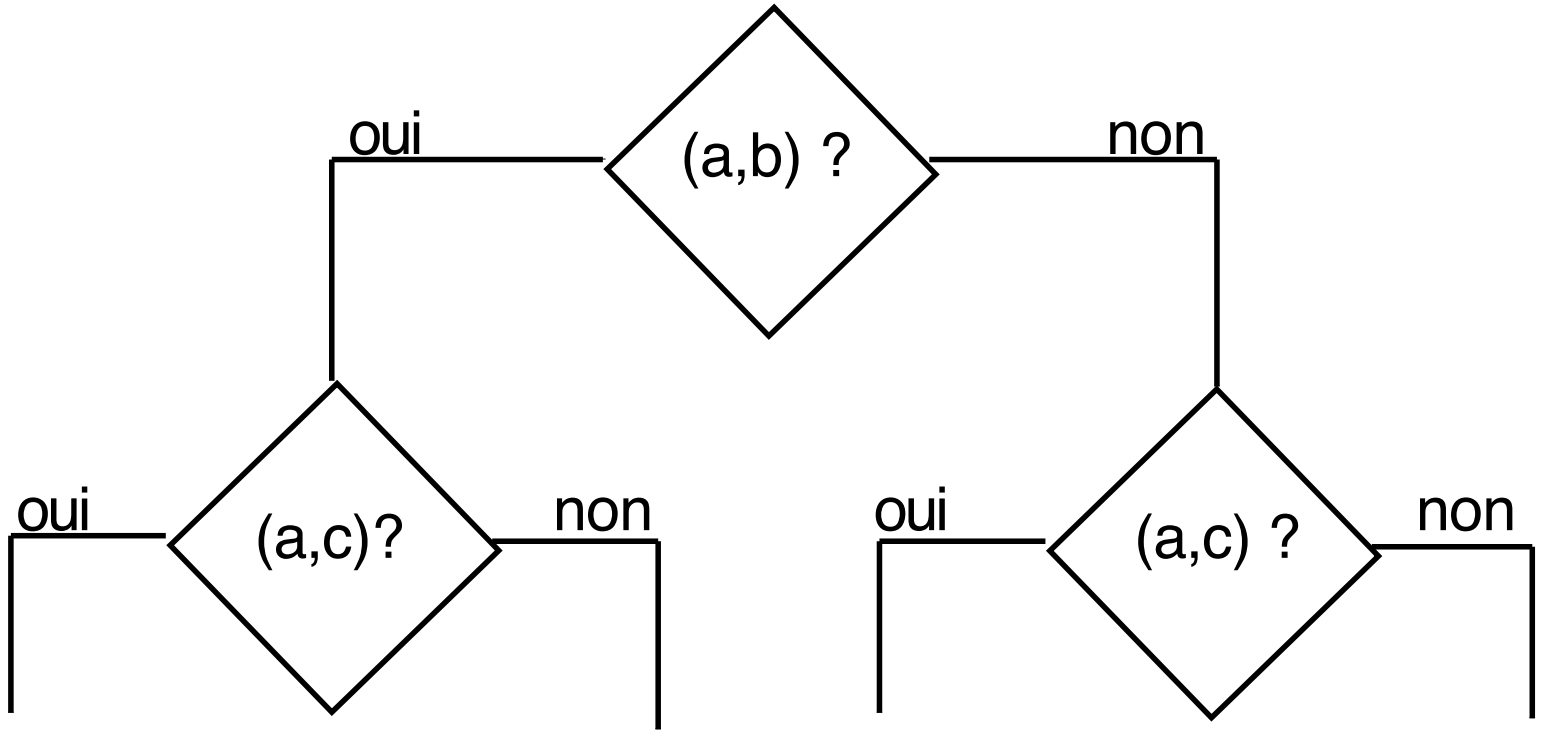
8.4 Branch-and-bound

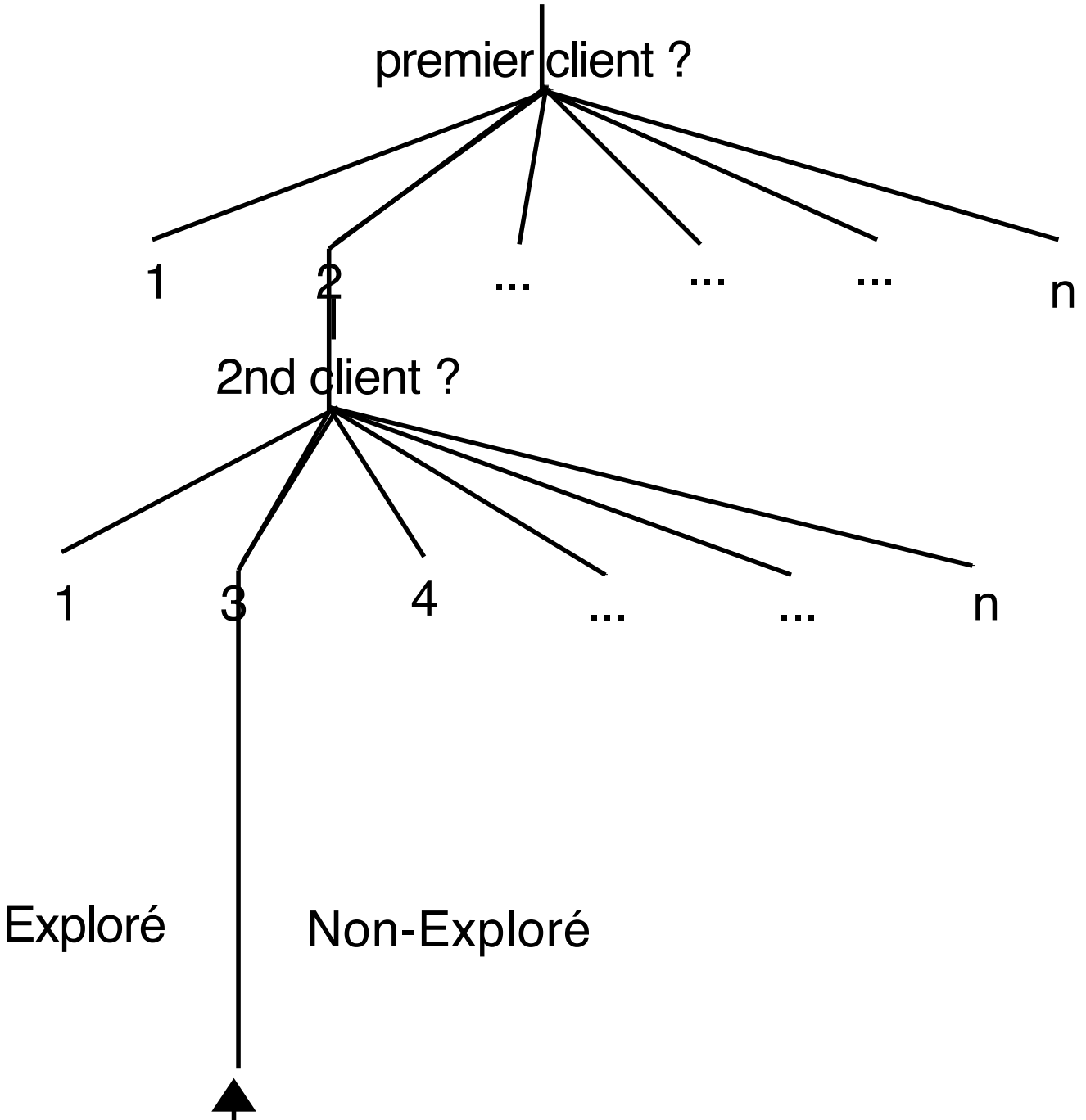
Amélioration : on utiliser des bornes (bound) sur le coût de la solution optimale parmi celles qu'on construira dans le sous-arbre d'exploration. Pour une minimisation, lorsque la borne inférieure d'une branche est supérieure à une solution déjà trouvée, ce n'est plus la peine de l'explorer : on l'élague (pruning).

On a donc intérêt à trouver rapidement une bonne solution.

8.5 Exemple : voyageur de commerce

On représente l'exploration par des noeuds de choix : inclure ou non l'arc (i, j) .




premier client ?
1
2
...
...
...
n
2nd client ?
1
3
4
...
...
n
Exploré
Non-Exploré

Borne inférieure

– Arc inclus $\Rightarrow$ coût connu.
– Chaque noeud doit avoir un arc entrant et un sortant : on complète avec les arc minimaux

– La demi-somme de ces arcs est une borne inférieure

Algorithme A^*

– Un choix est dit “ouvert” si on ne l’a pas exploré mais bien son père ;
– Heuristique : explorer le choix ouvert de moindre borne inférieure
– On élague lorsqu’une borne inférieure dépasse la meilleure valeur trouvée.