Algorithmique

2011-2012

Année Spéciale Informatique
ENSIMAG

Frédéric Devernay – Frederic.Devernay@inria.fr

Séance 1

Objectif du cours

(Algo 1 seulement)
Comprendre les mécanismes de base de la programmation : développement d'algorithmes efficaces, codage fiable et structuré, vérification et validation.
- Programmation en Ada
- Etude d'algorithmes fondamentaux, Etude de structures de données et algorithmes associés

Ensuite (Algo 2) :
- Programmation a objets en Java
- Modélisation et conception objet à l'aide de diagrammes de classes UML

Ouvrages de référence

Introduction à l'algorithmique, Thomas Cormen, Charles Leiserson, Ronald Rivest, Dunod, ISBN 2100039229.
Programmer en Ada95, John Barnes, Vuibert, ISBN 271178651X.

Charge de travail

Stage de rentrée (TD) : 21h (7 séances de 3h, séance 4 sur machines)
Cours : 16,5h (11 séances de 1h30)
TD : 16,5h (11 séances de 1h30)
TP : 16,5h (11 séances de 1h30)
Travail personnel (rapports de TP, révisions) : 20h
Stage C : 15h

Total Algo 1 : 21+49,5+20+15
Algo 1 + Algo 2 = 5 ECTS (soit environ 125h de travail au total)

Notation

ALGO 1 : règle de calcul : 1/6 (3 x note examen + 2 x note TP + note stage C)
Note TP : Qualité du code (5 points) + Qualité des tests (5 points) + Code fonctionne et passe mes tests (10 points)
ALGO 2 : règle de calcul : 1/3 (2 x note examen + Projet)

Note finale : 2/3 ALGOI + 1/3 ALGOII

Histoire des langages et de la programmation

Code machine, Fortran, Cobol, Lisp, C, PL1, Simula, Pascal, Ada 83 ; le mouvement objet : C++, Java

Code machine : manipulation directe des registres du processeur, pas de variables
Fortran (1954) et Autocode : **abstraction des expressions**, variables, tableaux, « `X = A + B(I)` »
Cobol (1960, mais en 2005, 75% des données du monde des affaires étaient traitées par des programmes en COBOL) : simplicité de programmation, calculs en virgule fixe adaptés à la comptabilité.
Algol 60 : **abstraction du contrôle**, les goto sont remplacés par if then else
Pascal : **abstraction des données** (types énumératifs, par ex. pour les couleurs d'un feu de signalisation)
C (1972) : programmation bas niveau (développement d'UNIX), gestion manuelle de la mémoire, aucun typage fort, d'où de nombreuses sources de bugs
Modula (dérivé de pascal) : **visibilité**, séparation interface/implantation
Ada 83 regroupe toutes ces formes d'abstraction. Réponse au cahier des charges (« Steelman document ») du DoD (Department of Defence) américain. Créé en France par CII Honeywell Bull.
C++ (années 80) : orienté objet, mais hérite des faiblesses de C, pas de typage suffisament fort pour une programmation sûre.
Java : comprend un langage, une API, et une JVM non standardisés ! Plus simple que C++ (pas de pointeurs), mais hérite de stupidités syntaxiques de C (=, {}, …)
Ada 95 : extension de types, polymorphisme dynamique, langage et API standardisés (mais il n'est pas nécessaire d'implanter toute l'API)

De Pacal à Ada : lecture et discussion du document

Pourquoi Ada ?
typage fort ➔ vérification statique (à la compilation, pas à l'exécution) ; exemple du new sur

```
type Metre is new float;
type Pied is new float;
  m : metre; p : pied; …
   p := pied(m) * 0.34 ; -- aurait évité que la sonde de la Nasa
                         -- s'écrase sur Mars !
```

La sonde spatiale Mars Climate Orbiter a été perdue le 23 septembre 1999. L'origine est un problème d'unité de mesure : « L'erreur d'unité concerne en fait des données d'accélération. Lockheed fournissait ses données en livres, une unité du système anglais, alors que les ingénieurs du JPL s'empressaient de les rentrer dans les ordinateurs en considérant que ces données représentaient des Newtons (unité du système métrique). »

Instruction conditionnelle

(if) et aiguillage (case)

Donner l'extrait du manuel de référence sur le case ; faire discuter et comprendre.

Exercice 1.1 : maximum de 3 nombres sous forme de fonction directement : algorithme efficace ?

Exemple du case : type jour et procédures *on_bosse, on_fait-la_fete, visite_médicale, ...*

Introduction aux invariants de boucle

1- initialisation
2- condition d'arrêt
3- propriété
4- corps
5- finalisation

(1)
while not (2) loop
  vérifier (3)
  (4)
end loop;
(5)

Exemple : somme de n nombres lus avec arrêt sur 0

```
somme := 0 ;
get(x) ;
while x <> 0 loop
  somme := somme + x ;
  get(x) ;
end loop ;
put(« la somme est : ») ;
put(somme) ;
new_line ;
```

Suite de Fibonacci

La célèbre suite de Fibonacci doit son nom au mathématicien italien Leonardo Pisano, mieux connu sous le pseudonyme de Fibonacci (1170 - 1250). Dans un problème récréatif posé dans un de ses ouvrages, le Liber Abaci, il décrit la croissance d'une population de lapins :

« Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ? »

Ce problème est à l'origine de la suite de Fibonacci, dont le n-ième terme correspond au nombre de paires de lapins au n-ème mois. Dans cette population (idéale), on suppose que :

* le premier mois, il y a juste une paire de lapereaux ;
* les lapereaux ne sont pubères qu'à partir du deuxième mois ;
* chaque mois, toute paire susceptible de procréer engendre effectivement une nouvelle paire de lapereaux ;
* les lapins ne meurent jamais (ie. la suite de Fibonacci est strictement croissante).

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

Exercice 1.2 : Ecrire l'algorithme de calcul de F(n) efficace (en utilisant une boucle, pas de récursivité)

Séance 2

Correction de Fibonacci en langage « algo ».

Itérations

Les différentes formes :

Intruction_loop ::= [idf_de_boucle :] [schema_iteration] **loop** suite_instruction **end loop** [idf_de_boucle]

schema_iteration ::= **while** condition | **for** identificateur **in [reverse]** définition_sous_type_discret

instruction_exit ::= **exit** [idf_de_boucle] [**when** condition]

Exemple à 2 sorties : somme et moyenne de nombres positifs, arrêt sur 0, et erreur si somme supérieure à 1000000. Deux possibilités : test de boucle sur somme (ci-dessous) ou sur le dernier entier lu.

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use  Ada.Text_IO, Ada.Integer_Text_IO ;

X : Positive ;
N : Positive ;
Somme : Positive ;
Moyenne : Float ;
begin
       -- initialisation : aucun nombre entré
       N := 0 ;
       Somme := 0 ;
       while Somme < 1000000 loop
              -- N est le nombre d'entiers positifs entrés
              -- Somme vaut la somme des entiers precedemment entres
              -- Moyenne vaut leur moyenne
              put("Entrez x : ") ;
              get(X) ;
              exit when X = 0 ;
              -- ajouter l'entier lu
              Somme := Somme + X ;
              N := N + 1 ;
              Moyenne := Float(Somme)/Float(N) ;
              -- afficher resultat partiel
              put("La somme vaut : ") ;
              put(Somme) ;
              new_line ;
              put("La moyenne vaut : ") ;
              put(Moyenne) ;
              new_line ;
       end loop ;
       -- finalisation
       put_line("Terminé") ;
end ;
```

Plus formellement : Principes fondamentaux et étapes de la construction d'une itération :

a- construire l'invariant de la boucle : description statique de l'état
b- construire les instructions qui dans le cas général permettent d'avancer (le corps de boucle)
c- condition(s) d'arrêt
d- finalisation si nécessaire : quel traitement de fin pour avoir le bon résultat
e- initialisation : comment entrer dans la boucle

Exercice : minimum, maximum, et donner la condition d'arrêt (sur n nombre, sauf si 0 apparaît).

Les types scalaires

(chapitre 6 du Barnes)

Déclarations

```
A : Integer;
B : Integer := 38;
Pi : constant Float := 3.14159_26536;
```

Pour les types numériques, on peut omettre le type dans la déclaration d'une constante

Blocs et portées

```
declare
  I : Integer := 0;
  K : Integer := I;
begin
  I := I+1;
end;
```

La portée des variables et constantes se limite au bloc. Sous-blocs. Occultation.

Types

« un ensemble de valeurs et un ensemble d'opérations primitives »
on ne peut affecter les valeurs d'un type à un autre type.

Sous-types

Sous-ensemble des valeurs d'un type défini au moyen d'une contrainte, mêmes opérations, et possibilité d'affecter des valeurs du type de base. Exception `Constraint_Error` en cas de violation de la contrainte.

```
subtype Jour_Du_Mois is Integer range 1..31;
J : Jour_Du_Mois;
I : Integer;
…
J := I; -- ok
```

Sous-types standards

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Qualification : Integer'(I) (lève éventuellement Constraint_Error, utilisation : ambiguïtés ou Positive'(I))

Types numériques

Integer (+range), opérations +,-,*,/,rem,mod,abs,**
Conversion : Integer(F : Float) arrondit au plus proche en s'éloignant de zéro
Priorités : = /= < <= > >=, + - (binaire), + - (unaire), * / mod rem, ** abs
Max et Min pour les types scalaires : Integer'Min(5,10)=5

Types énumératifs

```
type Couleur is (Rouge, Ambre, Vert);
type Roche is (Ambre, Beryl, Quartz);
```

pour distinguer les deux ambres au cas où il y aurait une ambiguïté, Couleur'(Ambre) ou Roche'(Ambre).
Contraintes à l'aide de range :

```
  subtype Jour_Ouvrable is Jour range Lun..Ven;
  J : Jour range Lun..Ven;
Couleur'Succ(Ambre)=Vert
T'Succ(T'Pred(X))=X
Couleur'Val(0) = Rouge
```

```
Couleur'Pos(Ambre)=1
T'Val(T'Pos(X)) = X
```
Possibilité de comparaisons : =, /=, <=, >, >=

Type booléens

```
type Boolean is (False, True);
```
opérations not, and, or, xor (priorité plus basse que tous les autres opérateurs), mais obligation de parenthèses pour les combiner : (B and C) or D

```
type Character :
type Character is (nul,…..,'0','1,'2',…,'A','B','C',..,'a','b',…) ; -- utilise
Latin-1
type Wide_Character ; -- sur 16 bits
```
donc ambiguïté de 'X' < 'L' : utiliser Character'('X') < 'L'

Divers

First, Last, Pos, Val, Succ, Pred s'appliquent à tous les types discrets
T'Image(a) retourne une chîne de caractères représentant a : put(T'Image(a))

Exercice 2.1 : convertir des caractères lus en séquence en un nombre (test d'arrêt : blanc, par de cas d'erreurs à ce stade) (utiliser if c in '0'..'9' then…)

Intervalles

expression_simple..expression_simple
exemples :
```
Lun..Jeu
Jour range Lun..Jeu
Jour_Ouvrable
Jour_Ouvrable range Lun..Jeu
```
(et aussi A'Range où A est un tableau)
utilisables dans instruction case à la suite de when, ou dans les tests « if n in intervalle »

Tableaux

```
A : array(Integer range 1..6) of Float;
```
on commence à l'indice qu'on veut (pas forcément 0 comme en C/C++/Java)
```
for I in 1..6 loop
  A(I) := 0;
end loop;
for I in A'Range loop
  A(I) := 0;
end loop;
AA: array(Integer range 0..2, Integer range 0..3) of Float:=((0.0, 0.0, 0.0, 0.0),
                                                             (0.0, 0.0, 0.0, 0.0),
                                                             (0.0, 0.0, 0.0, 0.0));
N : Integer := …;
B : array (Integer range 1..N) of Boolean;
J : Integer range A'Range; -- même contrainte que A
Heures_Ouvrées : array(Jour) of Float;
Demain : constant array(Jour) of Jour :=(Mar,Mer,Jeu,Ven, Sam,Dim,Lun);
```

Exercice 2.2 : recherche séquentielle d'un élément dans un tableau

Séance 3

```
type Vecteur_6 is array(1..6) of Float;
A : Vecteur_6;
B : Vecteur_6;
B :=A; -- recopie A dans B ;
```

```
C : array(1..6) of Float;
D : array(1..6) of Float;
C :=D; -- ILLEGAL
```

Tableaux non contraints

```
type Vecteur is array(Integer range <>) of Float; -- <> se dit « boîte »
V : Vecteur(1..5);
type Matrice is array(Integer range <>, Integer range <>) of Float;
subtype Matrice_3 is Matrice(1..3,1..3); -- obligation de donner toutes les bornes
```

les bornes sont évaluées au moment de la déclaration : peuvent être 1..N
si on initialise le tableau, les bornes commencent à Type'First. attention :

```
I2 : constant Matrice :=((1.0,0.0),(0.0,1.0)); -- borne inf Integer'First !!
```

'First, 'Last, 'Length, 'Range s'appliquent aux types ou aux variables tableaux.
Inpossible de déclarer des tableaux de tableau non contraints.

Agrégats de tableaux et leurs usages (affectation, initialisation en particulier)

```
I2 : constant Matrice := (1=> (1=>1.0, 2=>0.0), 2=>(1=>0.0,2=>1.0));
Jour_Ouvrable : constant array(Jour) of Boolean := (Lun..Ven =>True,
Sam|Dim=>False);
```

`others` peut apparaître en fin de liste d'agrégats par position ou par nom.
Impossible de mélanger agrégats par position et par nom.

Les chaînes

Un cas particulier de tableaux de caractères prédéfinis avec une bibliothèque d'opérateurs et fonction (c'est typiquement ce genre de bibliothèque (paquetage) qu'il faudra construire plus tard).

```
type String is array(Positive range <>) of Character;
AB : constant String := "A""B"; -- chaîne A"B
```

Extrait du RM sur le type string (extrait du manuel de référence)
On doit décider de la longueur de chaînes dans un tableau de chaînes (autre solution avec access plus tard) :

```
type Tableau_De_Chaines_6 is array(Positive range <>) of String(1..6);
```

Tranches

```
S : String(1..10);
S(1..4) := "BARA";
S(4..7) :=S(1..4); -- S(1..7) = "BARBARA"
T : array(0..100) of Integer;
U : array(0..3) of Integer;
U(0..3) := T(10..13); -- attention aux tailles !
```

Concaténation de tableaux et de chaînes

```
T := T(51..100) & T(1..50); -- échange des deux parties du tableau
S := "BAR" & 'B' & "ARA"; -- concaténation avec des éléments
```

Procédures et fonctions

Fonctions

```
-- function, désignateur, liste de paramètres, return, type de retour, is
function Sqrt(X : Float) return Float is
  R :Float; -- partie déclarative
begin
  -- calcul
  return R; -- valeur renvoyée
```

```
end Sqrt;
```
Liste de paramètres : type ou marque de sous-type (pas d'indication de sous-type par une contrainte explicite).
```
(I,J,K : Integer)
(Gauche :Integer ; Droite : Integer)
```
Déclaration (par exemple pour écrire deux fonctions qui s'appellent l'une l'autre) :
```
function Sqrt(X : Float) return Float ;
```
Appel de fonction : S := Sqrt(T+0.5) ; T+0.5 est évalué, Sqrt appelé, appel remplacé par valeur de R. Ordre d'évaluation des paramètres non défini.
Si pas de return atteint avant end, raise Program_Error.
Chaque appel crée un nouvel exemplaire des objets déclarés (y compris les paramètres). Ex. :
```
function Factorielle(N :Positive) return Positive is
  V : Positive ;
begin
  if N = 1 then
    V := 1 ;
  else
    V := N * Factorielle(N-1) ;
  end if ;
  return V ;
end Factorielle ;
```
Un paramètre peut être de type non contraint (bornes déduites du paramètre effectif, qui doit donc contenir des bornes, donc pas de others si agrégat) :
```
function Somme(A : Vecteur) return Float is … for I in A'Range loop … end
```

Définition d'opérateurs

```
function "*"(A,B :Vecteur) return Float ;
```

Procédures

```
procedure Ajouter(A,B : in Integer ; C : out Integer) ;
```
paramètres in, out, in out.
mêmes règles sur les paramètres formels/effectifs que pour les fonctions : les contraintes doivent être satisfaites en entrée pour in, en sortie pour out.
paramètres nommés (on ne peut utiliser others) :
```
Ajouter(C=>Q,A=>2+P,B=>37) ;
```
expressions par défaut dans une spécification de procédure :
```
procedure Kir(Base : Vin :=Muscadet ; Plus : Extra :=Sirop) ;
```

Principe fondamental de la construction des procédures (et fonctions) :

a- spécification de la procédure : quelles données, quels résultats (pas le comment, mais le quoi) : cette abstraction doit permettre le raisonnement modulaire.
b- une telle unité a une fonction logique la plus générale et bien identifiée, condition de sa réutilisabilité ; donc, sauf si c'est une procedure d'entrée/sortie, pas d'entrée/sortie (utiliser les exceptions pour signaler les erreurs);

Exercice 3.1 : Ecrire la procédure qui imprime toutes les combinaisons des lettres d'une chaîne. Preuve par induction de la correction de cette procédure.

Séance 4

Première prise en main sur machine

Rappel sur les procédures.

Recherche dichotomique

Algorithme, invariant de boucle, complexité, terminaison – attention, l'intervalle de recherche doit être strictement décroissant quel que soit le cas.

Puis TP en salle suivant le document : consultation du manuel en ligne Ada, étude de l'exemple d'un petit ensemble de programme de calcul matriciel, compilation, correction de code, mise au point, erreur d'exécution à analyser avec adastack ;
Exercice 4.1 : Insertion d'un code de recherche dichotomique dans un programme complet fourni par ailleurs.

Documents : la feuille d'instruction du TP et installation d'Ada sur PC.
en ligne : document sur différentes étapes, programme du calcul matriciel, programme où inserer la recherche dichotomique

Séance 5

Articles

Notations, intérêt, agrégat d'article

(Barnes 8.7)
(article comprenant trois composants)

```
type Date is
  record
    Jour : Integer range 1..31 ;
    Mois : Nom_De_Mois ;
    Annee : Integer ;
  end record
D : date ;
D.Jour := 14 ;
E : Date := (14, Juil, 1789) ;
E : Date ;
E :=D ;
E := (Annee => 1789, Mois => Juil, Jour => 14) ;
procedure Affiche_Date (D : Date) is
  begin
    Put(Image(D.Jour)&'/'&Image(D.mois)&'/'&Image(D.Annee)) ;
  end Affiche_Date ;
```

Possibilité de mettre des valeurs par défaut :

```
type Chaine is
      record
           Long : Natural := 0  ; -- longueur de la chaine que l'on considere
           Texte : String(1 .. Chaine_Max) ; -- les caracteres de la chaine
      end record ;
Chaine_Vide : constant Chaine := (Long => 0, Texte => (others => '.')) ;
```

Article avec partie discriminante

(Barnes 16)
Exemple sur les matrices carrées d'ordre N (cf. Barnes pp. 378) :

```
type Carree(Ordre : Positive) is
  record
    Mat : Matrice(1..Ordre, 1..Ordre) ;
  end record ;
M : Carree(3) ;
M : Carree(Ordre=>3) ;
M : Carree :=(3, (1..3 => (1..3 => 0.0))) ;
M : Carree(N) ;
M := (M.Ordre, (M.Mat'Range(1) => (M.Mat'Range(2) => 0.0))) ;
```

Dans ce cas, le discriminant est fixé à la déclaration de la variable, et ne peut être changé.
Si le discriminant a une valeur par défaut, il peut être changé par la suite par une affectation complète d'article.

```
type Carree(Ordre :Positive := 1)…
M : Carree ;
M = (2,(1..2 =>(1..2 => 0.0))) ;
M.Ordre := 4 ; -- Interdit !
```

Article avec partie variante:

```
type Sexe is (Masculin, Feminin) ;
type Individu (Genre : Sexe) is
  record
    --  partie fixe
    -- …
    case Sexe is
      when Masculin => A_Barbe : Boolean ;
      when Feminin => C_Cheveux : Couleur ;
         Nb_Enfants : Natural ;
    end case ;
  end record ;
subtype Homme is Individu(Genre => Masculin) ;
..
P : Individu := (Masculin, …);
H : Individu(Masculin) ;
```

`P.Genre` est mutable (peut être changé), contrairement à `H.Genre` qui est une contrainte.

Paquetages

(Barnes 11)
Notation, séparation entre spécification et implantation :
Spécification (fichier idf_package.ads) :
[<clauses_with_use>]
[<parametres_generiques>]
package <idf package> is
  <declarations>
  [private <declarations>]
end <idf_package>
Corps (fichier idf_package.adb) :
[<clauses_with_use>]
package body <idf_package> is
<declarations_internes>
<corps_de_l_implementation>
end <idf_package>
Exemple avec les nombres complexes et les deux représentations (partie réelle, partie imaginaire, ou module et argument) : intérêt : l'utilisateur ne voit pas le changement de l'implantation, transparence, portabilité, evolutivité sont permis :

```
package Nombres_Complexes is
  type Complexe is private ;
  I : constant Complexe ;
  function « + » (X : Complexe) return Complexe ;
  …
private
  type Complexe is
    record
      Rl, Im : Float ;
    end record ;
  I : constant Complexe := (0.0, 1.0) ;
end ;
```

on peut changer la partie privée sans impact sur l'utilisateur en :

```
private
  Pi : constant := 3.14159_26536 ;
```

```
  type Complexe is
    record
      R : Float ;
      Theta : Float range 0.0..2.0*Pi ;
    end record ;
  I : constant Complexe := (1.0, 0.5*Pi) ;
end ;
```

Private : protection artificielle
Limited private : pas de = (ou /=) ni de := ; intérêt : l'égalité de pile dans des tableaux : on ne teste que l'égalité sous le sommet de pile et non tous les élements. pas de := signifie aussi pas de valeur initiale !

Paquetages génériques

(Barnes 17)
Paramétrage des paquetages par types (Barnes 17.2), constantes – par ex. longueur max de chaînes -- , procédures (Barnes 17.3) – par ex. relation d'ordre.

```
generic
   type T is private ;
   type Indice is (<>) ; -- ceci est type discret
   type D(<>) is private ; -- type de discriminant inconnu(String,Matrice)
   valeur : integer ;
   with function Truc (X, Y : Indice) return Boolean ;  -- ne pas oublier
le with !!!
   type Vecteur is array(Indice range <>) of T ;
package Tentative is
  …
end Tentative ;

usage :
package Ma_Tentative is new Tentative (Boolean, Positive, String, 30, ">",
tableau-bit);
  use Ma_Tentative;
```

Réflexion : qu'implique une déclaration d'un parametre type qui est private ? (réponse : selon les besoins, passer aussi en paramètre du paquetage des fonctions de comparaison, de recopie des objets, etc . cf. Barnes 17.3).

Exceptions

Exemple des matrices de la séance précédente :

```
function Trace(M : Matrice) return Float is
  Somme : Float := 0.0 ;
begin
  if M'First(1) /= M'First(2) or M'Last(1) /= M'Last(2) then
    raise Non_Carree ;
  end if ; …
```

Syntaxe et sémantique des exceptions (par l'exemple) :

```
package Pile is
  Erreur_Pile : exception ;
  procedure Empiler(X :Integer) ;
  function Depiler return Integer ;
end Pile ;

package body Pile is
  Max : constant := 100 ;
  S : array(1..Max) of Integer ;
  Sommet : Integer range 0..Max := 0;
  procedure Empiler(X : Integer) is
  begin
    if Sommet = Max then
```

```
      raise Erreur_Pile ; -- evite Constraint_Error a la ligne suivante
    end if ;
    Sommet := Sommet + 1 ;
    S(Sommet) := X ;
  end Empiler
  function Depiler return Integer is
  begin
    if Sommet = 0 then
      raise Erreur_Pile ;
    end if ;
    Sommet := Sommet - 1 ;
    return S(Sommet+1) ;
  end Depiler

begin
  -- instructions
  -- …
exception
  when Storage_Error =>
    Put(« Plus d'espace memoire ») ;
    ..
  when Erreur_Pile =>
    Put(« manipulation incorrecte de la pile ») ;
    Nettoyer ;
    raise Autre_Erreur ;
  when others =>
    Put(« autre erreur ») ;
    Nettyoyer ;
    raise ;
end ;
```

Exceptions courantes

(Barnes 14.1)

- de base : `Program_Error ; Storage_Error ; Constraint_Error` ;
- E/S : `Data_Error ; End_Error, Name_Error, …`

paramètre de choix pour afficher des informations sur une exception inconnue:

```
when Event: others =>
  Put("exception inattendue");
  Put(Exception_Name(Event));
  New_Line;
  Put(Exception_Message(Event));
  Nettoyer;
  raise;
end;
```

Exercice 5.1 : Ecriture en Ada d'un paquetage de lecture avec reprise en cas d'erreur de saisie (lire numéro de code trois fois max, et invalidation si non numérique).

Problème de l'écriture d'un paquetage de chaine de longueur variable et redéfinition de la concaténation (préparation du premier TP) :
Type = implantation + constructeurs + testeur + sélecteur + modifieur
Application pour les chaînes de longueurs variables:
Constructeur : string2chaine, concaténation (extension de la concatenation de string), chaine_vide ;
Testeur : Est_Vide, Est_???? est mon nom favori pour les tests
Sélecteur : longueur chaine-> entier ; texte : chaine -> string
Modifieur : add_char_to_string ....
Exemple du paquetage Chaines (TP1) :

```
-- Paquetage pour les chaines de caracteres de longueur bornee
-- Specification
```

```
package Chaines is

   -- Taille maximale des chaines considerees
   Chaine_Max : constant Natural := 80 ;

   -- Le type des chaines de caracteres de longueur bornee
   type Chaine is private ;

   -- La chaine vide
   Chaine_Vide : constant Chaine ;

   -- Exception levee en cas de debordement (creation d'une chaine dont la
   -- longueur depasse Chaine_Max)
   Erreur_Chaine : exception ;

   -- Creation_Chaine(S) est la chaine correspondant a S
   -- En cas de debordement, l'exception Erreur_Chaine est levee
   function Creation_Chaine(S : String) return Chaine ;

   -- Acces_String(C) est la string correspondant a C
   function Acces_String(C : Chaine) return String ;

   -- Acces_Longueur(C) est la longueur de la chaine C
   function Acces_Longueur(C : Chaine) return Natural ;

   -- Majuscule(C) passe la chaine C en majuscules
   -- (les caracteres autres que 'a' .. 'z' ne sont pas modifies)
   procedure Majuscule(C : in out Chaine) ;

   -- C1 = C2 est vrai ssi les deux chaines C1 et C2 ont les memes caracteres
   function "="(C1, C2 : Chaine) return Boolean ;

   -- C1 < C2 est vrai ssi la chaine C1 est strictement avant la chaine C2
   -- (ordre lexicographique)
   function "<"(C1, C2 : Chaine) return Boolean ;

   -- Ajouter_Caractere(C, X) ajoute le caractere X au bout de la chaine C
   -- En cas de debordement, l'exception Erreur_Chaine est levee
   procedure Ajouter_Caractere(C : in out Chaine ; X : in Character) ;

   -- Ajouter_Chaine(A, B) ajoute la chaine B au bout de la chaine A
   -- En cas de debordement, l'exception Erreur_Chaine est levee
   procedure Ajouter_Chaine(A : in out Chaine ; B : in Chaine) ;

   -- C1 & C2 est la concatenation des chaines C1 et C2
   -- En cas de debordement, l'exception Erreur_Chaine est levee
   function "&"(C1, C2 : in Chaine) return Chaine ;

   -- Put(C) affiche la chaine C
   procedure Put(C : in Chaine) ;

   -- Put_Line(C) affiche la chaine C suivie d'un retour chariot
   procedure Put_Line(C : in Chaine) ;

   -- Get_Line(C) lit une chaine C sur l'entree standard
   procedure Get_Line(C : out Chaine) ;

private
   -- definition du type Chaine
   type Chaine is
      record
           -- Long :  longueur de la chaine que l'on considere
           Long : Natural := 0  ;
           -- Texte : les caracteres de la chaine
           Texte : String(1 .. Chaine_Max) ;
      end record ;

   -- definition de la chaine vide
```

```
   Chaine_Vide : constant Chaine := (Long => 0, Texte => (others => '.')) ;
end Chaines ;
```

Documents :
Version papier de la correction de l'exercice « paquetage de lecture »

Séance 6

Documents :
paquetage E/S ;
IO-exception (qui contient aussi la semantique du get_line)

Entrées-sorties

(Barnes 20.5) Les E/S ne font pas partie du langage, mais des paquetages standard ont été définis dans le RM95.
Entrées-sorties binaires :
I : Integer := 75 ;
Write(F,I) ;
Entrées-sorties texte :
Put(F,I) ;

Entrées-sorties **sequentielles** (Barnes 20.5, package Ada.Sequential_IO) : un fichier est ouvert soit en lecture (In_File), soit en écriture (Out_File), soit en écriture à la fin du fichier (Append_File). **Opérations** Create, Open, Close, Delete, Reset, Read, Write ; **Testeurs** Mode, Name, Form, Is_Open.
principe du séquentiel : à la queue leuleu … intérêt : rapidité.

Entrées-sorties **directes** (package Ada.Direct_IO, ne seront pas examinées par la suite) : ajoutent le mode Inout_file, pas de Append_File. L'indice courant peut être changé par Set_Index, et Read et Write permettent de lire/écrire à un indice donné. Testeurs Index et Size.
Généralités sur les E/S : binaire versus texte

Entrées-sorties de texte (Barnes 20.6): standard, consultation par utilisateur (mais pas économique en place, et lecture par un programme plus coûteuse)

Présentation des E/S, des fichiers ; des fonctions afférentes (renvoi au manuel de référence, impression d'une partie du manuel à étudier) et des exceptions ;
Exercice de lecture : lire le get_line dans le RM95, section A.10.7, p332.

Exercice 6.1 : Fusion de deux tableaux ou de deux fichiers

1- Construire l'algorithme de fusion de deux tableaux (d'entiers) triés
2- Comment généraliser à la fusion de deux entités (tableaux, fichiers, listes…) pour lesquelles seule une opération d'entrée (lire la valeur de l'élément courant - élément du tableau, ligne de fichier… - et passer à l'élément suivant) et une opération de sortie (écrire élément dans tableau ou dans fichier) sont possibles. Quel est l'intérêt de cette solution ?
3- On souhaite écrire un paquetage générique permettant de fusionner toutes les entités : quels doivent être les paramètres génériques ?

Étude du fichier paquetage p_fusion fourni en exemple

Document : programme d'interclassement de fichier (p_fusion)

Séance 7

Structures chaînées

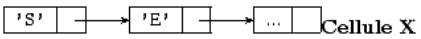
On a vu comment représenter des listes (suites) d'éléments sous forme de tableaux.
Problème : l'insertion d'un élément en tête ou au milieu de la liste requiert un décalage de tous les éléments, coûteux en place.
Concrètement : remplacer un mot par un mot de longueur différente dans une chaîne de caractères.

Solution (Liste simplement chaînée):

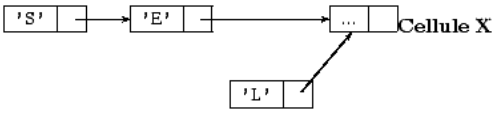
```
type Cellule is record Lettre: Character range 'A'..'Z'; Suiv: Natural; end record;
Tas: array(1..TailleMemoire) of Cellule ;
```

Tas(K) : première lettre, Tas(Tas(K.Suiv)) : deuxième lettre, etc. Si Suiv=0, fin de la liste.
Suiv est un **pointeur** vers l'élément suivant de la liste.
Nécessite de gérer une pile de cases vides…

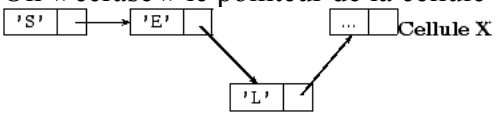
représentation sous forme de dessin (transformation de 'SE' en 'SEL':



Création d'une nouvelle cellule contenant 'L', recopie du pointeur vers X:



On « écrase » le pointeur de la cellule 'E' par un pointeur vers 'L'



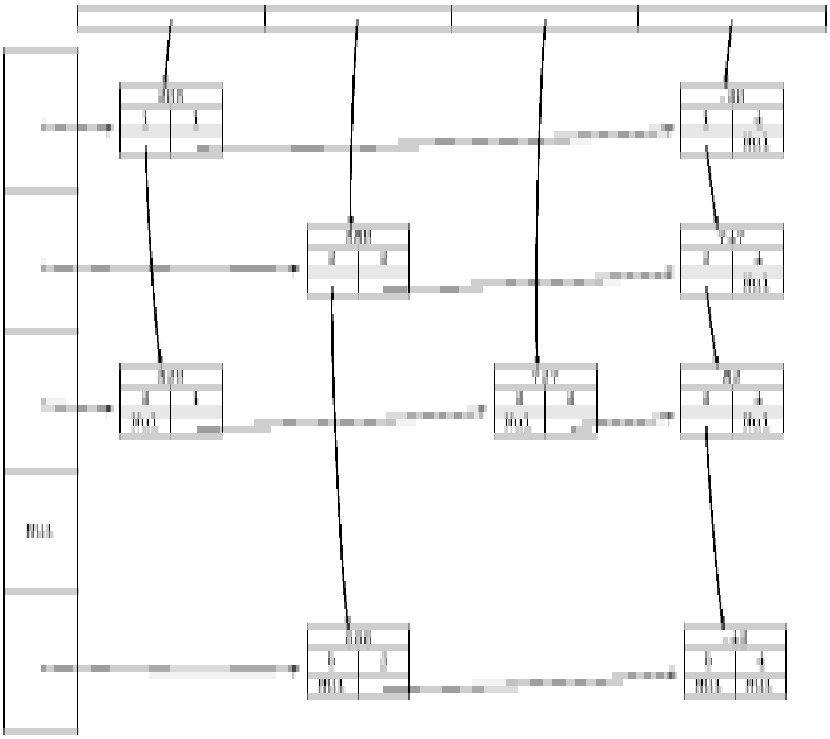
Listes doublement chaînées : permettent le parcours dans les deux sens :

```
type Cellule is record
  Lettre: Character range 'A'..'Z';
  Pred, Suiv: Pointeur
end record ;
type DListe is record
  Tete,Queue: Pointeur ;
end record ;
```

Exercice 7.1: insertion dans une liste doublement chaînée

Matrice creuse :

```
type Cellule is record
  Coef: Integer ;
  NumL,NumC: Positive ;
  SuivC, SuivL: Pointeur ;
end record ;
type Matrice is record
  TLig: array(1..5) of Pointeur ;
  TCol: array(1..4) of Pointeur ;
end record ;
```



Le type access et le constructeur new.

Aussi appelés pointeurs ou références dans d'autres langages.
En Ada :

```
type Cellule ;
type Ptr_Cellule is access Cellule ;

subtype Majuscule is Character range 'A'..'Z' ;

type Cellule is record
   Lettre: Majuscule;
   Suiv: Ptr_Cellule ;
end record ;
```

Initialisation et allocation

```
T : Ptr_Cellule ; -- initialisé à null par défaut
T:=new Cellule'((Lettre => E, Suiv => T)) ;
```

ou

```
T:=new Cellule ;
T.all := (Lettre => 'A', Suiv => T) ;
```

Storage_Error si pas assez de place mémoire.

Déréférencement et allocation, avec « all »

```
T.all.Lettre := ... ; -- vérifie que T n'est pas null, sinon Constraint_Error
T.all.Suiv := ... ;
```

ou

```
T.Lettre := ... ;
T.Suiv := ... ;
```

Libération

En général pas nécessaire (le langage repose sur un « garbage collector ») et dangereuse (Ada ne vérifie pas si on accède à un pointeur qui a été libéré)
Utilise la procédure générique Ada.Unchecked_Deallocation :

```
generic
  type Objet is limited private ;
```

```
  type Name is access Object ;
procedure Ada.Unchecked_Deallocation(X : in out Name) ;
```
utilisation :
```
procedure Liberer is
  new Ada.Unchecked_Deallocation(Cellule, Ptr_Cellule) ;
```

Attention

Problèmes d'aliasing :
```
T, U : Ptr_Cellule ;
…
T := U ;
T.all := (Lettre => 'A', Suiv => T) ; -- modifie aussi U.all, même zone mémoire
```
Passage de paramètres : lorsqu'un access est passé en paramètre in, la zone pointée peut être modifiée

Implantation d'une liste chaînée

Construction de base, exemple d'utilisation simple avec des access :
```
with Ada.Text_IO, Ada.Integer_Text_Io ;
use Ada.Text_IO, Ada.Integer_Text_Io ;

procedure ExAcces is

   type Cellule ;
   type Ptr_Cellule is access Cellule ;

   subtype Majuscule is Character range 'A'..'Z' ;

   type Cellule is record
      Lettre: Majuscule;
      Suiv: Ptr_Cellule ;
   end record ;

   -- Affichage d'une liste chaînée
   procedure Put(L: Ptr_Cellule) is
      -- procédure d'affichage
      Cour: Ptr_Cellule := L ;
   begin
      while Cour /= null loop
         Put(Cour.Lettre) ;
         Cour:=Cour.Suiv ;
      end loop ;
      New_Line ;
   end ;

   -- Insertion dans une liste chaînée
   procedure InsereAvant (T: in out Ptr_Cellule;
                          E: in Majuscule) is
      -- requiert: T est la tete (éventuellement nulle)
      -- de la liste.
      -- insere un nouvel élément avant T
   begin
      T:=new Cellule'((Lettre => E, Suiv => T)) ;
   end ;

   procedure InsereApres(Q: in out Ptr_Cellule;
                         E: in Majuscule) is
      -- requiert: Q est en element (non null) de la liste.
      -- insere un nouvel élément apres Q
      -- pour insérer en queue de liste, passer Q=queue de liste
   begin
      Q.Suiv:=new Cellule'((Lettre => E,Suiv => Q.suiv)) ;
      Q:=Q.Suiv ;
   end ;
```

```
   -- Construction d'une liste à partir d'une String
   -- (on suppose ici que celle-ci ne contient que des
   --  majuscules).
   function Construit1(S: String) return Ptr_Cellule is
      -- méthode par insertions en tête
      R: Ptr_Cellule :=  null ;
   begin
      for I in reverse S'Range loop
         InsereAvant(R,S(I)) ;
      end loop ;
      return R ;
   end ;

   function Construit2(S: String) return Ptr_Cellule is
      -- méthode par insertions en queue
      R,Q: Ptr_Cellule ;
   begin
      if S'Length = 0 then
         return null ;
      end if ;
      R:=new Cellule'((Lettre => S(S'First),Suiv => null)) ;
      -- R reste sur la tete
      -- alors que Q va rester sur la queue.
      Q:=R ;
      for I in S'First+1..S'Last loop
         InsereApres(Q,S(I)) ;
      end loop ;
      return R ;
   end ;

   -- Transformation des "SE" en "SEL"
   procedure ChangeSE_SEL(L:Ptr_Cellule) is
      Courant: Ptr_Cellule := L ;
      LuS : Boolean := False ;
      X: Ptr_Cellule ;
   begin
      while Courant /= null loop
         -- LuS est true si le précédent caractère est un S
         X := Courant.Suiv ;
         case Courant.Lettre is
            when 'S' => LuS := True ;
            when 'E' =>
               if LuS then
                  Courant.Suiv := new Cellule'((Lettre => 'L',
                                                Suiv => X)) ;
                  LuS:=False ;
               end if ;
            when others =>
               LuS:=False ;
         end case ;
         Courant:= X ;
      end loop ;
   end ;


   L: Ptr_Cellule ;
begin
   L:=Construit2("CSSECSESE") ;
   Put(L) ;
   ChangeSE_SEL(L) ;
   Put(L) ;
end ;
```

Cours 8

Les listes et algorithmes classiques : parcours, insertion, liste triée et insertion, suppression d'un élément. Version chaînée, version consécutive (tableau) : comparer les opérations pour les deux implémentations.

Astuce de l'élément factice dans une listé triée simplifiant l'insertion :
Une liste chaînée vide peut être représentée
- par un pointeur null
- par un élément factice ne contenant aucune valeur. Cet élément factice permet d'éviter de traiter l'insertion dans une liste vide comme un cas particulier (on fait toujours pointer le « suivant » d'un élément de la liste vers l'élément nouvellement créé). Son champ « suivant » est null si la liste est vide, sinon c'est un pointeur vers le premier élément. Se souvenir de la méthode de construction par insertion en queue de la séance précédente : on a du traiter le premier élément séparément.

Complexité de la recherche d'un élément

Liste non triée : recherche en O(n)

Recherche auto-adaptative :
But : réorganiser la liste au fur et à mesure des recherches afin de minimiser le nombre de comparaisons lors des futures recherches.

Idée 1 : après chaque recherche, placer l'élément en tête de liste. Méthode bien adaptée à l'implémentation chaînée, pas du tout à l'implémentation contiguë.

Idée 2 : après chaque recherche, faire progresser l'élément d'une place vers la tête de liste. Méthode adaptée aux deux implémentations.

Analyse de ces deux idées (Knuth) : si le nombre de recherche sur une liste de n éléments est inférieur ou égal à n*n , l'idée 1 est plus performante. Au delà de n*n, c'est l'idée 2 la meilleure. (En effet, l'idée 2 est meilleure, mais dans l'idée 1 les éléments atteignent plus rapidement un état stationnaire)

Liste triée : recherche séquentielle identique au cas non trié, sauf qu'on s'arrête dès que l'élément courant est supérieur à l'élément cherché. en O(n) aussi, donc.
Recherche dichotomique possible uniquement dans l'implémentation consécutive/contiguë, en O(log2(n)). Nous verrons d'autres structures dynamiques (arbres binaires de recherche) plus adaptées à la recherche d'éléments.

Renverse de liste sur place

(complétée en TD)
transformer
A->B->C->D->E->F
en
F->E->D->C->B->A

Le problème est a priori difficile. Pour le résoudre, il faut se placer "au milieu" de l'execution, donner l'invariant de boucle, puis construire l'itération, et enfin condition d'arret, initialisation, finalisation.

« au milieu », on a:

```
A<-B<-C  D->E->F
       ^  ^
  debut  fin
```

et il faut avoir en fin d'iteration:

```
A<-B<-C<-D  E->F
          ^  ^
     debut  fin
```

Exercice 8.1 : Ecrire l'itération de boucle pour la renverse de liste sur place

Bilan

liste triée et non triée, consécutive ou chaînée, selon les opérations : appartenance, insertion/suppression, min, union/intersection, parcours.

Listes spéciales

liste circulaire (pour faire un parcours complet, comparer le pointeur courant avec le pointeur vers le premier élément parcouru), listes doublement chaînées, superposition de listes permettant de gérer plusieurs classements simultanés (lignes/colonnes pour une matrice, nom/prénom, etc.).

Exercice 8.2 : on a une fonction (voir Ada.Numerics.Discrete_Random)

```
function Aleat (K : Natural) return Natural ;
--  retourne un nombre au hasard entre 0 et K, bornes comprises.
```

On veut imprimer une permutation aléatoire des nombres de 0 à N. Comment représenter E, l'ensemble des nombres qu'il reste à imprimer ?

Note : Cette méthode permetrait aussi de tirer k éléments parmi n, mais n'est pas très efficace lorsque k<<n, à cause de son utilisation mémoire.

Note : méthode permettant de tirer k éléments parmi n en O(n) (Knuth 3.4.2) – c'est un problème différent (on s'intéresse à tirer une combinaison et non une permutation) mais la solution est élégante (efficace lorsque k est proche de n, sinon utiliser l'algorithme RANKSB qui est en O(k) lorsque k << n, cf. pp 38-9 de Nijenhuis & Wilk (1975) Combinatorial Algorithms, Academic Press.):

```
// Choose k numbers from a list of n numbers (ranging from 0 - n-1)
for (i = 0; i < n && k > 0; ++i)
{
       if (random(n-i) < k)
       {
              trace(i);
              k--;
       }
}
```

Solution complète intégrant les deux algorithmes, avec choix automatique : http://devernay.free.fr/vision/src/deal.c

Cours 9

Mesures de complexité

temps d'exécution, nombre d'opérations (affectations, comparaisons, opérations arithmétiques), taille mémoire utilisée.

Algo itératif : $C(n) = \sum_{i=0}^{n} f(i) \approx \int_{1}^{n+1} f(t)dt$

Algo récursif : $C(n) = a.C\left(\frac{n}{K}\right) + f(n)$ (cours suivant)

Complexité dans le meilleur cas : $f_{\min}(n) = \min\{f(d) / |d| = n\}$

Complexité dans le pire cas : $f_{\max}(n) = \max\{f(d) / |d| = n\}$

Complexité en moyenne :

$f_{moyenne}(n) = moyenne\{f(d) / |d| = n\} = \sum_{k \geq 0} k.p(C(n) = k) = \sum_{k \geq 0} p(C(n) \geq k)$

Complexité en moyenne avec hypothèse d'équiprobabilité (toutes les données de taille n ont la même probabilité d'occurrence) : $f_{moyenne}(n) = \frac{\sum_{|d|=n} f(d)}{\text{nb de données de taille n}}$

Ex : recherche séquentielle, $f_{moyenne}(n) = \sum_{i=1}^{n} i.p(C(n) = i) = \sum_{i=1}^{n} i.\frac{1}{n} = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$

Exercice 9.1 : Un calcul de maximum.
La fonction suivante, écrite en Ada, calcule l'élément maximal d'un vecteur T . L'appel initial est ValMax(T,K) où K est l'indice du dernier élément de T :

```
type Tableau is array(Integer range <>) of Float;

function ValMax (T: Tableau, I: Integer) return Float is
begin
   if I = T'First then
      return T(T'First);
   elsif T(I) > ValMax(T, I-1) then
      return T(I);
   else
      return ValMax(T, I-1);
   end if;
end ValMax;
```

Un utilisateur, qui teste cette fonction sur des vecteurs de tailles croissantes, vous dit : « je ne comprends pas, la fonction marche jusqu'à 27 éléments, mais semble boucler pour 28 ».
Comment interprétez-vous ce comportement ?
Que faut-il faire pour y remédier ?

Calcul des 2 éléments maximaux d'un vecteur

Exercice 9.2 : On désire trouver les deux éléments les plus grands d'un tableau T. On supposera que les données sont équidistribuées. Comparer la complexité des deux versions Algo1 et Algo2 ci-dessous réalisant ce calcul. Que peut-on en conclure ?

```
type Tableau is array(Integer range <>) of Float;

procedure Algo1(X : Tableau; Max1, Max2 : out Float) is
   Aux : Float;
begin
```

```
      Max1 := X(X'Last);
      Max2 := X(X'Last - 1);
      if Max1 < Max2 then
        Aux := Max1 ;
        Max1 := Max2 ;
        Max2 := Aux ;
      end if ;
      for I in reverse X'First..(X'Last-2) loop
        if X(I) >= Max1 then
           Max2 := Max1;
           Max1 := X(I);
        elsif X(I) > Max2 then
           Max2 := X(I);
        end if;
      end loop;
   end Algo1;
```

La seconde version Algo2 consiste à remplacer la partie intérieure de la boucle par :

```
        if X(I) >= Max2 then
           if X(I) >= Max1 then
              Max2 := Max1;
              Max1 := X(I);
           else
              Max2 := X(I);
           end if;
        end if;
```

```
   procedure Algo2(X : Tableau; Max1, Max2 : out Float) is
      Aux : Float;
   begin
      Max1 := X(X'Last);
      Max2 := X(X'Last - 1);
      if Max1 < Max2 then
        Aux := Max1 ;
        Max1 := Max2 ;
        Max2 := Aux ;
      end if ;
      for I in reverse X'First..(X'Last-2) loop
        if X(I) >= Max2 then
           if X(I) >= Max1 then
              Max2 := Max1;
              Max1 := X(I);
           else
              Max2 := X(I);
           end if;
        end if;
      end loop;
   end Algo2;
```

Cours 10

Définition de O(f(n)) :

∃ A, ∃ α ∀n>A $g(n) < \alpha f(n)$ => g is said to be $O(f(n))$

Examples :

n^2 is $O(n^3)$ (why?), $1000n + 10^{10}$ is $O(n)$

Borne inférieure sur la complexité du tri : n! résultats/conclusions possibles, donc si tests binaire (comparaisons), on obtient un un arbre de hauteur minimale Log_2 n! ~ log_2 (n^n / e^n) [formule de stirling, n ! ~ sqrt(2*pi*n)*(n/e)^n] ~ O(n log n)

Quel résultat, quelle donnée : liste → liste

Transparents « algos de tri », présentation de tri sélection, insertion, quicksort, mergesort

Récurrence sur données, sur résultats :

A ce stade :

- tri par sélection : récurrence sur le résultat,
- tri par insertion, récurrence simple sur les données

- tri fusion : récurrence dichotomique sur les données
Théorême de la complexité de diviser pour régner document : poly d'algo chapitre 12
Powerpoint « time complexity » à partir de la page 12
Application au tri fusion : complexité en O(n ln n)

Récursivité

Classe d'algorithmes récursifs :

Algorithmes sur des structures récursives (listes, en référence en avant arbres)
Algorithmes liés à une définition récursive (Fibonnacci, Hanoi, énumération combinatoire)

Exemple d'algorithme récursif pour résoudre un problème a priori difficile : Les tours de Hanoï

```
procedure Hanoi (n:integer; i, j: integer);
begin
  if n > 0 then
    begin
    Hanoi (n-1, i, 6-(i+j));
    writeln (i:2, '->', j:2);
    Hanoi (n-1, 6-(i+j), j);
    end;
end;
```

Cours 11

Preuve et terminaison

Terminaison

Les appels imbriqués terminent-ils? Typiquement, si on ne prend pas de précaution dans un graphe avec circuit, on ne termine pas, et donc la récurrence est invalide mathématiquement.

Preuve

La **preuve** c'est vraiment la récurrence : c'est vrai quand la condition d'arrêt est valide, supposons par hypothèse de récurrence que c'est vrai pour les appels internes, et prouvons que c'est vrai pour la procédure en entier.
Exemple de preuve classique: à la fin de l'appel récursif, la donnée globale est restituée dans l'état qu'elle avait avant l'appel.

Indécidabilité de la terminaison d'une fonction récursive

Il est impossible de créer un programme permettant de prouver si une fonction récursive termine. Exemple donné par l'élève Gödel au maître Hilbert :
Ne laissez plus boucler vos programmes! Utilisez notre fonction Termine(u). Elle prend comme paramètre le nom de votre procédure et donne pour résultat true si la procédure ne boucle pas indéfiniment et false sinon. En n'utilisant que les procédures pour lesquelles Termine répond true, vous évitez tous les problèmes de non terminaison. D'ailleurs, voici quelques exemples:

```
procedure F;
  var x: integer;
  begin
    x := 1;
    writeln(x);
  end;
procedure G;
  var x: integer;
  begin
```

```
    x := 1;
    while (x > 0) do x:= x + 1;
  end;
```
pour lesquels Termine(F) = true et Termine(G) = false.

Mais considérons la procédure suivante :
```
procedure Absurde;
  begin
    while Termine(Absurde) do
    ;
  end;
```
Si la procédure Absurde boucle indéfiniment, alors Termine(Absurde) = false. Donc Absurde termine.
Sinon, si la procédure Absurde ne boucle pas indéfiniment, alors Termine(Absurde) = true. et Absurde boucle indéfiniment. Il y a donc une contradiction
sur les valeurs possibles de Termine(Absurde).

Exemples de preuves

recherche récursive dans une liste (if L = null return false ; if E = L.value return true else return recherche(E,L.suiv) ;), parcours d'un arbre binaire, Fibonacci (cours 1), Hanoi (cours 10), énumération des n! suites ordonnées de lettres (cours 3)

Les n-reines

Problème (Max Bezzel, 1848) : placer n reines sur un échiquier nxn sans qu'elles ne puissent se prendre
Algorithme naïf (tester toutes les possibilités) en O((n^2)^n) !!!
Solution (7.1.3 du poly): Placer une reine sur chaque ligne i en vérifiant que la position est libre. Si aucune position n'est libre sur la ligne i, revenir à la ligne i-1 et changer la position de la reine sur cette ligne : « backtracking ». On réalise une exploration arborescente de toutes les solutions, en se souvenant des solutions explorées précédemment.
Optimisations : utiliser les symétries (par exemple, n'explorer que les (n+1)/2 premières cases de la première ligne).
Heuristiques : pour obtenir une (et non toutes les solutions) le plus vite possible, on utilise deux types d'heuristiques :
- « first fail principle » = explorer d'abord les choix qui donnent le moins de solutions (profondeur faible de l'arbre de recherche)
- on explore l'abord les solutions donnant le plus de possibilités (le plus de cases vides dans le cas des reines). on risque d'y passer du temps si l'heuristique n'est pas parfaite, sinon on réussit au premier coup…

Les arbres

transparents « Arbres »
Annexe A pour les définitions.
parcours en largeur et en profondeur

retour sur les n reines, implantation :
```
subtype pos_etendu is natural range 0..n ;
subtype pos is pos_etendu range 1..n ;
ligne : array(pos) of pos_etendu := (others => 0) ;

procedure Reine(i : pos_etendu) is
-- entree : i reines sont placées, mémorisées dans le tableau ligne
```

```
-- au retour, toutes les solutions complétant cette solution partielle
--   sont imprimées
begin
  if i=n then
    -- toutes les reines sont placées, on est à une feuille de l'arbre de recherche
    imprimer(ligne) ;
    return ;
  end if
  -- exploration des différentes possibilités
  k := i+1 ;
  for j in 1..n loop
    if position_libre (k,j) then
      ajouter_une_reine (k,j) ;
      reine(k) ;
      enlever_une_reine(k,j) ;
    end if ;
  end loop ;
end Reine ;

procedure ajouter_une_reine(k,j :pos) is
begin
  ligne(k) := j ;
end ajouter_une_reine ;

procedure retirer_une_reine(k,j :pos) is
begin
  ligne(k) := j ;
end retirer_une_reine ;

-- programme principal :
reine(0) ;
```

Tables de hachage

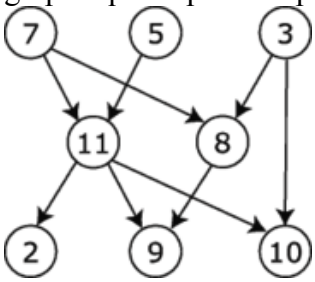
En TD :
Présentation de la structure de donnée "table de hachage"
Algorithmes de recherche / ajout / suppression dans une table de hachage

Cours 12

Rappel sur les arbres : Algo de parcours en largeur d'abord, parcours avec gestion de priorité.

Graphes

(Chapitre 6 du poly)
G=(sommets=X, edges=E)
non-orienté : (x,y) dans E implique (y,x) dans E
DAG = graphe acyclique orienté (en anglais directed acyclic graph ou DAG) identifie un graphe qui ne possède pas de cycle, et dont les arcs sont orientés :



Représentations possibles pour un graphe

Matrice d'ajacence

```
type sommet is natural range 1..N ;
type graphe is array (sommet,sommet) of boolean ou integer;
G : graphe ;
```
- occupation mémoire en O(n^2), non efficace si G non dense, efficace si G presque complet
- accès direct aux arcs, successeurs et prédécesseurs

Table des successeurs

```
type arc_elem ;
type arcs is access arc_elem ;
type arc_elem is
  record
    extremite : sommet ;
    cout : integer ;
    suivant :arcs ; end record ;
type les_successeurs is record
  nbre :integer := 0 ;
  tete : arcs := null ; end record ;
type table_successeurs is array (positive range) of les_successeurs ;
```
- on peut ajouter la liste des prédécesseurs
- correspond à une représentation de la matrice d'adjacence sous forme de matrice creuse (cf. cours 7)

Représentation dynamique : liste chaînée de sommets

pour chaque sommet liste des successeurs et des prédécesseurs (nécessaires pour le retrait d'un sommet)

Algorithmes pour les graphes

Exploration arborescente de tous les sommets identique a l'exploration des arbres, mais :
- on peut accéder au même sommet par plusieurs chemins
- il peut y avoir des circuits -> bouclage de l'algo

* Donc marquer les sommets déjà rencontrés (tableau de booleens) pour parcourir une seule fois chaque sommet. Complexité O(n).

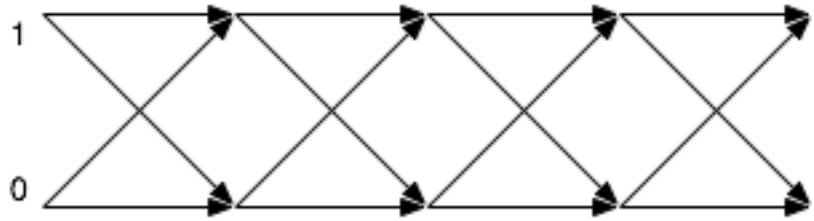
Construction de tous les chemins :

* Si on veut construire tous les chemins elementaires (ne contenant pas deux fois le même sommet), il faut tester si un sommet appartient au chemin considéré (exploration en profondeur d'abord)

```
type sommet is natural range 1..N ;
type chemin is record
  les_points : array(sommet) of sommet ;
  indice_dernier : (0..N) := 0 ;
  dans_chemin : array(sommet) of boolean := (others => false) ;
end record ;

ch : chemin ;

procedure Explore(extremite : sommet) is
-- entree : ch contient un chemin partiel, indice_dernier=extremite
-- sortie : construit tous les chemins elementaires prolongeant ch
begin
pour chaque y successeur de extremite loop
  if not ch.dans_chemin(y) then
    mettre_dans_ch(y) ;
    Explore(y) ;
    retirer_de_ch(y) ;
  end if ;
end loop ;
```

```
end Explore ;

sommet depart := 1 ;
ch.indice_dernier := 1;
ch.les_sommets(1) := depart ;
Explore(depart) ;

procedure Explore_Jusqua(extremite : sommet, arrivee : sommet) is
-- entree : ch contient un chemin partiel, indice_dernier=extremite
-- sortie : affiche tous les chemins elementaires prolongeant ch jusqu'a arrivee
begin
pour chaque y successeur de extremite loop
  if not ch.dans_chemin(y) then
    mettre_dans_ch(y);
    if y <> x then
       Explore(y) ;
    else
       affiche_chemin;
    end if
    retirer_de_ch(y);
  end if ;
end loop ;
end Explore ;
```

Complexité de l'algo d'exploration :
Dans le cas d'un arbre, O(n)
Graphe ci-dessous : O(2^(n/2))=O(2^n) (nombres binaires de taille n/2)



Dans le cas d'un graphe complet ? O(n !)
* Pour explorer les chemins non-élémentaires (où on autorise les boucles), adopter une solution adhoc selon le problème traité pour éviter les boucles infinies (par exemple marquer les arcs et non les sommets)

Chemins optimaux

chaque arc a un coût, et on cherche le chemin optimal de a à b

Solution par exploration en profondeur d'abord (depth-first)

mettre à jour une solution (optimale) contenant un chemin de a à b et son coût. on ne parcourt un arc que si le coût total est inferieur a optimal.cout, et on met à jour optimal si on arrive à b :

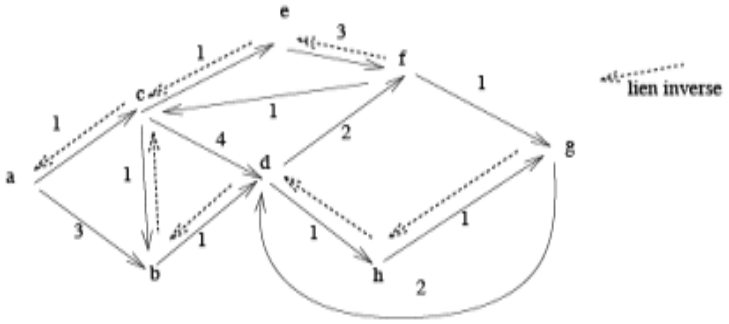
```
 procedure Meilleur(extremite : sommet, alpha : integer) is
-- entree : ch contient un chemin partiel, indice_dernier=extremite
--          alpha : cout total du chemin, avec alpha < optimal.cout
-- sortie : construit tous les chemins elementaires prolongeant ch
begin
if extremite = b then
  optimal.ch := ch ;
  optimal.cout := alpha ;
  return ;
end if ;
pour chaque y successeur de extremite loop
  if not ch.dans_chemin(y) and cout(x,y)+alpha < optimal.cout then
    mettre_dans_ch(y) ;
    Meilleur(y,alpha+cout(x,y)) ;
    retirer_de_ch(y) ;
```

```
  end if ;
end loop ;
end Meilleur ;
```

Non-optimal : si on arrive en un sommet y avec un coût supérieur à un chemin déjà parcouru arrivant à y, inutile de continuer !
Complexité dans le cas pire en O(n !)

Solution par exploration en largeur d'abord (breadth-first programmation dynamique)

tout chemin optimal de a à b passant par y commence par le chemin optimal de a à y.
Il suffit de conserver en chaque sommet le chemin optimal permettant d'y arriver et le coût associé.
Il suffit même de conserver le prédécesseur et de reconstruire le chemin optimal par *backtracking* !



On fait un parcours en largeur d'abord, ce qui permet de calculer tous les chemins optimaux de taille k à chaque « itération » et d'éliminer au plus vite les chemins non-optimaux.
La complexité dans le cas pire est encore en O(n !) si à chaque nouveau passage en y on a un nouveau chemin optimal.
Heuristique : parcourir d'abord les arcs de coût faible parmi les successeurs.

Algorithme de Dijkstra (coûts positif ou nuls)

si on a atteint un sommet x avec un coût minimal parmi tous les sommets en cours d'examen, le chemin de a à x ne pourra plus être remis en cause, car aucun chemin optimal vers ce sommet ne pourra passer par les autres successeurs.
On modifie le parcours en largeur d'abord pour traiter en priorité le successeur qui a le coût optimal :

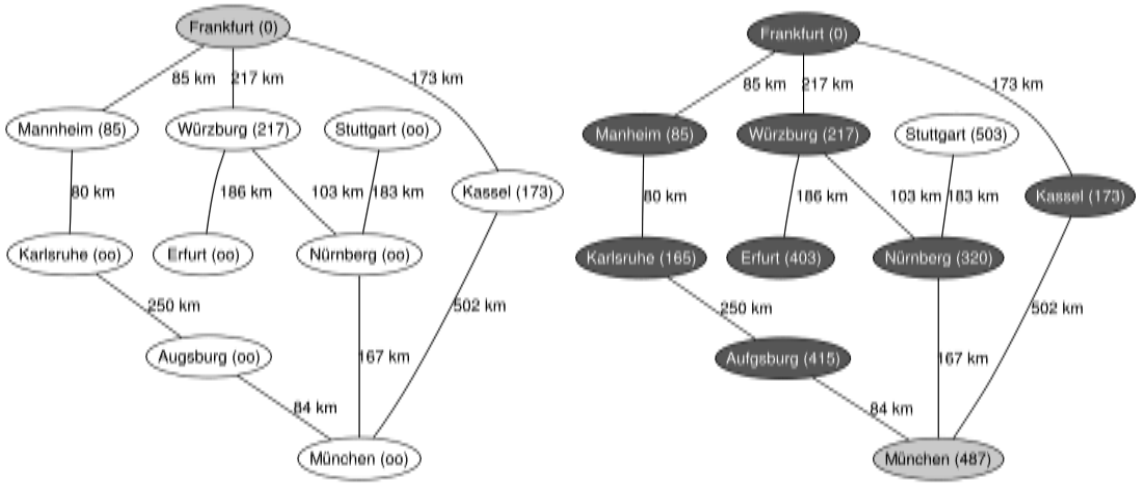
```
Attente := {depart};
lien_inverse(tous_les_sommets) := lien_vide;
distance(depart) := 0;
distance(autres) := infini;
tant que Attente non vide repeter
  oter-minimum(Attente, x) ; -- retire le x tq distance(x) minimum
  -- (oter-tete pour le parcours en largeur d'abord)
  si x = arrivee, c'est fini, sinon :
  pour y dans successeur de x repeter
    si distance(x)+cout(x,y) < distance(y) alors
      distance(y) := distance(x)+cout(x ,y) ;
      lien_inverse(y) := x ;
      inserer(Attente, y) ; -- inserer-queue pour le parcours en largeur d'abord
    fsi
  fin pour
ftq
```

Backtracking :

```
X := arrivée;
chemin := { x };
tant que lien_inverse(x) != lien_vide répéter
```

```
  x := lien_inverse(x);
  inserer-tete(chemin, x);
ftq
si x = départ alors OK, sinon y n'est pas atteignable depuis x
```

Pour chaque sommet, ensemble des successeurs examiné une fois et une seule, donc complexité O(m*alpha+n*beta) (m arcs, n sommets, alpha = coût d'insertion dans Attente, beta=coût d'extraction du minimum)
On utilise pour cela une file de priorité, utilisant un tas (heap), alpha=cte*ln(n), beta=cte*ln(n), ce qui donne C=O((m+n)log(n)) (avec un tas de Fibonacci, O(m+nlog(n)))
Exemple :



Cours 13

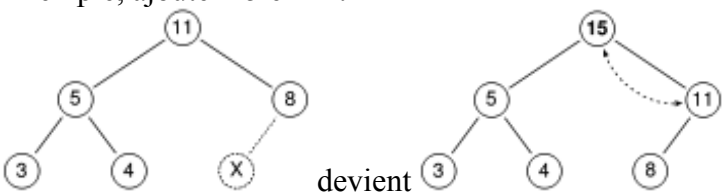
Rappel de Dijkstra, d'où l'utilité de la file de priorité

Tas binaire

C'est un arbre binaire respectant les contraintes suivantes :
- propriété de forme : c'est un arbre parfait (dernier niveau plein ou si le dernier niveau n'est pas plein les nœuds sont remplis de gauche à droite)
- propriété de tas : c'est un arbre partiellement ordonné : la valeur en tout nœud est supérieure ou égale à celle de ses fils

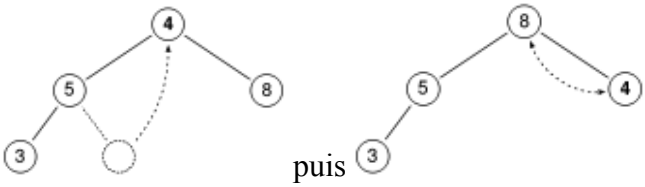
Ajout d'un élément

en bas du tas, puis échange avec le père jusqu'à rétablir la propriété de tas. O(logn)
Exemple, ajouter 15 en X :



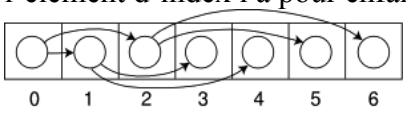
Retrait de l'élément max

le remplacer par le dernier élément, puis rétablir la propriété de tas en échangeant avec les fils qui lui sont supérieurs. O(logn)



Implantation

Utilisation d'un tableau : pas de place perdue avec les pointeurs
l'élément d'index i a pour enfants 2i+1 et 2i+2 et pour parent floor((i-1)/2)



Application : Heapsort

On traite le tableau comme un tas non ordonné, puis on fait « remonter » les éléments, du dernier au premier. On extrait ensuite un à un les éléments pour les placer à la fin du tableau. Complexité totale O(nlogn), mais nécessite beaucoup d'échanges. Pas de mémoire supplémentaire nécessaire.

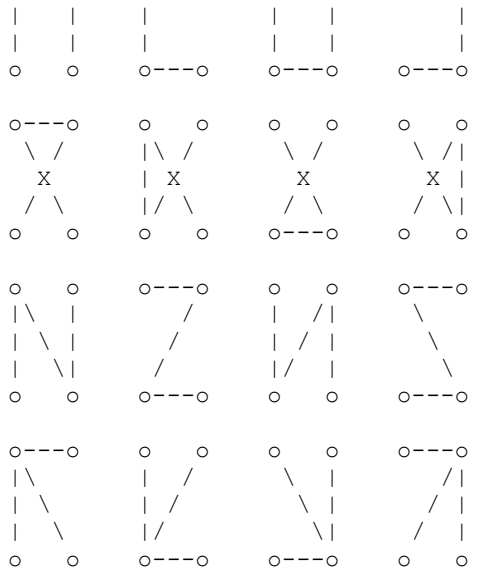
Composantes connexes (graphe non-orienté)

```
procedure test-connected(G)
   {
   choose a vertex x
   make a list L of vertices reachable from x,
   and another list K of vertices to be explored.
   initially, L = K = x.

   while K is nonempty
      find and remove some vertex y in K
      for each edge (y,z)
         if (z is not in L)
         add z to both L and K

   if L has fewer than n items
      return disconnected
   else return connected
   }
```

Arbre de recouvrement minimal (minimum spanning tree)

Voir http://www.ics.uci.edu/~eppstein/161/960206.html .
Arbre de recouvrement : contient tous les sommets d'un graphe et est un arbre. Exemple :

a 16 arbres de recouvrement:



Lorsque les arrêtes ont des coûts (ou des longueurs), les arbres ont évidemment des longueurs différentes, comment trouver l'arbre de recouvrement minimal ?

Pourquoi les MST ?
- design d'un réseau téléphonique
- solution approximative du problème du voyageur de commerce (travelling salesman problem – TSP), mais le MST n'est pas un chemin, il faut donc parcourir deux fois chaque arrête. Le coût du MST est inférieur au TSP (car l'ensemble des chemins est inclus dans l'ensemble des arbres de recouvrement, 12 des 16 spanning trees ci-dessus sont des chemins), mais le coût du TSP est moins de deux fois le coût du MST.

Comment trouver le MST ?

Méthode stupide : calculer le coût de tous les arbres, mais complexité exponentielle !
Supposons pour la suite qu'il existe un unique MST.
Alors nous allons utiliser une propriété permettant de construire le MST une arrête à la fois :
Lemme : Soit X un sous-ensemble des sommets de G, et soit l'arrête e la plus petite arrête reliant X à G-X. Alors e fait partie du MST.
Preuve : Supposons qu'on ait un arbre de recouvrement T ne contenant pas e, prouvons que T n'est pas le MST. Soit e=(u,v), avec u dans X et v hors de X. Alors T étant un arbre de recouvrement contient un chemin unique de u à v, qui lorsqu'on ajoute e forme un cycle dans G. Ce chemin contient donc nécessairement une autre arrête joignant X à G-X. T+e-f est un autre arbre de recouvrement (même nombre d'arrêtes, et complètement connecté puisque tout chemin passant par f peut être remplacé par un chemin passant par e). Il a un coût plus faible puisque e a un coût plus faible que f. Donc T n'était pas minimal, CQFD.

Algorithme de Kruskal

We'll start with Kruskal's algorithm, which is easiest to understand and probably the best one for solving problems by hand.

Kruskal's algorithm:
sort the edges of G in increasing order by length
keep a subgraph S of G, initially empty
for each edge e in sorted order
  if the endpoints of e are disconnected in S

```
    add e to S
return S
```

Note that, whenever you add an edge (u,v), it's always the smallest connecting the part of S reachable from u with the rest of G, so by the lemma it must be part of the MST.

This algorithm is known as a **greedy algorithm**, because it chooses at each step the cheapest edge to add to S. You should be very careful when trying to use greedy algorithms to solve other problems, since it usually doesn't work. E.g. if you want to find a shortest path from a to b, it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property we proved.

Analysis: The line testing whether two endpoints are disconnected looks like it should be slow (linear time per iteration, or O(mn) total). But actually there are some complicated data structures that let us perform each test in close to constant time; this is known as the union-find problem. The slowest part turns out to be the sorting step, which takes O(m log n) time.

Algorithme de Prim

Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time.

```
Prim's algorithm:
let T be a single vertex x
while (T has fewer than n vertices)
{
    find the smallest edge connecting T to G-T
    add it to T
}
```

Since each edge added is the smallest connecting T to G-T, the lemma we proved shows that we only add edges that should be part of the MST.

Again, it looks like the loop has a slow step in it. But again, some data structures can be used to speed this up. The idea is to use a heap to remember, for each vertex, the smallest edge connecting T with that vertex.

```
Prim with heaps (heap elements are indexed by vertex):
make a heap of values (vertex,edge,weight(edge)) ; edge is the edge connecting to parent
    initially (v,-,infinity) for each vertex
    let tree T be empty
replace (r,-,infinity) with (r,-,0) ; so that root is the first extracted vertex
while (T has fewer than n vertices)
{
    let (v,e,weight(e)) have the smallest weight in the heap
    remove (v,e,weight(e)) from the heap
    add v and e to T
    for each edge f=(u,v) ; i.e. for each u adjacent of v
    if u is not already in T ; use a boolean array is_in_T[]
        find value (u,g,weight(g)) in heap
        if weight(f) < weight(g)
```

```
        replace (u,g,weight(g)) with (u,f,weight(f)) ; use an array[vertex] –>index in heap
    }
```

Analysis: We perform n steps in which we remove the smallest element in the heap, and at most 2m steps in which we examine an edge f=(u,v). For each of those steps, we might replace a value on the heap, reducing it's weight. (You also have to find the right value on the heap, but that can be done easily enough by keeping a pointer from the vertices to the corresponding values.) I haven't described how to reduce the weight of an element of a binary heap, but it's easy to do in O(log n) time. Alternately by using a more complicated data structure known as a Fibonacci heap, you can reduce the weight of an element in constant time. The result is a total time bound of O(m + n log n).

Algorithme de Boruvka

Inventé pour optimiser le réseau électrique de la Moravie en 1926.
(Actually Boruvka should be spelled with a small raised circle accent over the "u".) Although this seems a little complicated to explain, it's probably the easiest one for computer implementation since it doesn't require any complicated data structures. The idea is to do steps like Prim's algorithm, in parallel all over the graph at the same time.

```
Boruvka's algorithm:
make a list L of n trees, each a single vertex
while (L has more than one tree)
   for each T in L, find the smallest edge connecting T to G-T
   add all those edges to the MST
   (causing pairs of trees in L to merge)
```

As we saw in Prim's algorithm, each edge you add must be part of the MST, so it must be ok to add them all at once.

Analysis: This is similar to merge sort. Each pass reduces the number of trees by a factor of two, so there are O(log n) passes. Each pass takes time O(m) (first figure out which tree each vertex is in, then for each edge test whether it connects two trees and is better than the ones seen before for the trees on either endpoint) so the total is O(m log n).

Un algorithme hybride

This isn't really a separate algorithm, but you can combine two of the classical algorithms and do better than either one alone. The idea is to do O(log log n) passes of Boruvka's algorithm, then switch to Prim's algorithm. Prim's algorithm then builds one large tree by connecting it with the small trees in the list L built by Boruvka's algorithm, keeping a heap which stores, for each tree in L, the best edge that can be used to connect it to the large tree. Alternately, you can think of collapsing the trees found by Boruvka's algorithm into "supervertices" and running Prim's algorithm on the resulting smaller graph. The point is that this reduces the number of remove min operations in the heap used by Prim's algorithm, to equal the number of trees left in L after Boruvka's algorithm, which is O(n / log n).

Analysis: O(m log log n) for the first part, O(m + (n/log n) log n) = O(m + n) for the second, so O(m log log n) total.

Cours 14

Programmation dynamique

Détail de la programmation dynamique, transparents Dynamic Programming
Tri topologique du graphe pour la mise en œuvre

Cours 15

Branch and bound (séparer et évaluer)

Transparents Branch and Bound
Branch and bound ; principe général (exploration et évaluation) (chapitre 15)

Problème du sac à dos

Le problème du sac à dos, noté également KP (en anglais, Knapsack Problem) est un problème d'optimisation combinatoire. Il modélise une situation analogue au remplissage d'un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.
Stratégie d'évaluation.
Les 2 façons d'opérer la séparation : explore (Volume atteint, objets_à_considérer) avec récurrence sur les objets et séparation sur le nombre d'objets au rang considéré ; ou bien séparation sur le rang.

Problème des cageots de fraises

Voici un autre problème, dont le cas général peut s'aborder avec le branch and bound comme par la programmation dynamique.
Nous nous intéressons à la distribution de n cageots de fraises dans p magasins. Les bénéfices que l'on peut retirer de chaque magasin est fonction du nombre de cageots fourni. Ainsi, $b_j(i)$ représente le bénéfice que tire le magasin j de la vente de i cageots ($1 \leq i \leq n$ et $1 \leq j \leq p$).
Nous appelons gains marginaux les gains supplémentaires obtenus par cageots : $g_j(i) = b_j(i) - b_j(i-1)$, avec $b_j(0) = 0$ (aucun bénéfice pour la vente de 0 cageots).
1. Si, pour chaque magasin j, la fonction de gains marginaux g_j est décroissante, proposer un algorithme optimal. Donner un contre-exemple si les gains marginaux ne sont pas décroissants.
2. Combien de solutions au problème existent dans le cas général? Comb(p+n-1,n) ?
3. Proposer un algorithme qui calcule le coût optimal pour répartir les cageots de fraises dans les magasins pour le cas général. Quelle est sa complexité ?

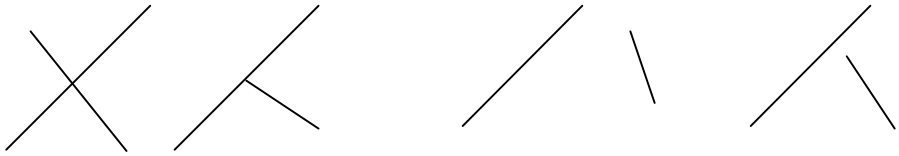
Cours 16

Algorithmes géométriques

Algorithmes géométriques (Sedgewick, chapitre 24) = Géométrie algorithmique

Intersection de deux segments

point : (x,y)
segment : (point,point)
polygone : array of point
fonction « intersection » qui retourne true si deux segments s'intersectent :



- solution1, calculer l'intersection des deux droites support, et vérifier qu'elle est à l'intérieur des deux segments
- solution2, utilise la fonction ccw (counter-clockwise) qui retourne 1 si quand on parcours les points dans le sens inverse des aiguilles d'une montre, -1 si on les parcours dans le sens inverse. Cas spéciaux (points alignés : -1 si p0 entre p2 et p1, 0 si p2 est entre p0 et p1, 1 si p1 entre p0 et p2 (simplifie les algorithmes suivants) :

```
int ccw(struct point p0,
	struct point p1,
	struct point p2)
{
	float dx1, dx2, dy1, dy2 ;
	dx1 = p1.x – p0.x ; dy1 = p1.y – p0.y ;
	dx2 = p2.x – p0.x ; dy2 = p2.y – p0.y ;
	if (dx1*dy2 > dy1*dx2) return +1 ; // pente du segment 1= dy1/dx1
	if (dx1*dy2 < dy1*dx2) return -1 ;
	// trois points colinéaires
	if ((dx1*dx2 < 0) || (dy1*dy2 <0)) return -1 ; // p0 entre p1 et p2
	if ((dx1*dx2+dy1*dy1) < (dx2*dx2+dy2*dy2)) return +1 ; // p0-p1-p2
	return 0 ;
}
```

fonction intersect : si les extremites de chaque segment sont des deux côtés de l'autre segment, alors ils s'intersectent

```
int intersect(struct line l1, struct line l2)
{
	return ((ccw(l1.p1, l1.p2, l2.p1) * ccw(l1.p1, l1.p2, l2.p2)) <= 0)
		&& ((ccw(l2.p1, l2.p2, l1.p1) * ccw(l2.p1, l2.p2, l1.p2)) <= 0) ;
}
```

cette solution paraît compliquée, mais c'est la seule qui marche dans tous les cas (4 pts collinéaires par exemple)

Point à l'intérieur d'un polygone

Déterminer si un point est à l'intérieur d'un polygône :
prendre un segment partant du point allant « loin » dans la direction x
compter le nombre d'intersections avec le polygône : pair = point à l'extérieur, impair = point à l'intérieur
MAIS si le segment « test » passe exactement par des sommets, que faire ?
solution : parcourir le polygône, et incrémenter un compteur à chaque fois qu'on passe d'un côté à l'autre du segment test. Ne rien faire si on tombe dessus :

```
int inside(struct point t, struct point p[], int N)
{
	int i, count = 0, j = 0 ;
	struct line lt,lp ;
	p[0] = p[N] ; p[N+1] = p[1] ;
	lt.p1 = t ; lt.p2 = t ; lt.p2.x = FLT_BIG ;
	for (i=1 ; i<=N ; i++)
	{
		lp.p1 = p[i] ; lp.p2 = p[i];
		if ( !intersect(lp,lt))
		{
			lp.p2 = p[j] ; j = i ;
			if (intersect(lp,lt)) count++ ;
		}
	}
```

```
        return count & 1 ;
}
```

Enveloppe convexe d'un ensemble de points Q

le plus petit polygône convexe pour lequel chaque point de Q est soit sur la frontière, soit à l'intérieur.
analogie de l'élastique autour des clous
Algo de la marche de Jarvis (package wrapping) :
- partir d'un point dont on est sûr qu'il est sur l'enveloppe convexe : point d'ordonnée minimum
- trier les sergments reliant ce point à tous les autres par leur angle avec l'axe des abscisses. Celui qui a le plus petit angle donne le point suivant.
- l'algo termine quand on retombe sur le premier point
Complexité : O(n*n) au pire, O(n*h) en réalité
Ressemble au tri sélection : faire un tableau de points, à chaque nouveau point de l'enveloppe, l'échanger avec l'élément k du tableau, où k est le nombre de points de l'enveloppe en cours de construiction. Les points à tester sont alors celui d'indice 1, et ceux d'indice k+1..n

Algo Graham scan :
- prendre le point de départ.
- trier tous les autres points par angle croissant avec ce point
- il en résulte un polygône non convexe
parcourir ce polygône, et tant qu'on tourne à gauche, ajouter le point.
Si on tourne à droite, éliminer le point précédent, jusqu'à ce qu'on tourne à gauche
Complexité : tri O(nlogn) + au pire deux parcours de l'ensemble (pour ajouter les points + pour les éliminer) O(2n)
=> O(nlogn)

QuickHull (enveloppe rapide):
on part de deux points extremes L et R, et on sépare l'ensemble en deux selon LR. On lance ensuite la procedure recursive suivante sur chaque moitié :

```
Quickhull( PointSet S, Point L, Point R )
1. if ( S = {L, R} )  // if the only points in the set are L and R
2. return {L, R}
3. else
4. H = farthest( S, L, R )
5. S1 = points of S on or left of LH
6. S2 = points of S on or right of HR
7. return ( Quickhull(S1, L, H) □ Quickhull( S2, H, R ) )
where farthest( PointSet S, Point L, Point R) is a function that determines which
point
yields the triangle of largest area.
```

Complexité moyenne O(nlogn), pire O(n2)

Diviser pour régner (Preparata et Hong)
- séparer en deux par une droite
- calculer l'enveloppe convexe de chaque moitié
- « fusionner » les deux enveloppes convexes en trouvant les deux segments à ajouter
Complexité O(nlogn)

Question : lesquels de ces algos passent en 3D ?
Pas le Graham scan. Package wrapping possible. Quickhull OK. Divide and conquer toujours ! (même en N-dimensions)

Autres algos « classiques » de géométrie algorithmique : Voronoï (taches de giraffe), Delaunay (dual de Voronoï, largement utilisé en mécanique et en simulation numérique)…

Cours 17

Révision des notions essentielles

Boucles et invariant
Récursivité
Objets abstraits et implantation
Représentation des ensembles (tableau de synthèse)
Algo sur les graphes

Cours 18

Recherche de chaînes

(basé sur le chapitre 19 de Sedgewick, Algorithms in C)
chaîne de texte de longueur N, motif de longueur M
brute-force :

```
int brutesearch(char *p, char *a)
{
  int i, j, M = strlen(p), N = strlen(a) ;
  for (i=0 , j=0 ; j<M && i<N ; i++,j++)
    while (a[i] != p[j]) {
      i -= j-1 ;
      j = 0 ;
    }
  if (j == M)
    return i-M ;
  else
    return i ;
}
```

Figure 19.1 de Sedgewick, Algorithms in C :

Figure 19.1 Brute-force string search in binary text.

Complexité ? MxN

Algorithme de Knuth-Morris-Pratt

Si le motif ne correspond pas, le « faux départ » contient des caractères qu'on connaît à l'avance, on peut donc « reculer » le pointeur i moins loin.
Exemple : motif=10000000, si on a un faux départ, aucun des j-1 caractères précédents ne peut être un départ, on peut donc remplacer i -= j-1 par i++, complexité N
Mais ceci ne se généralise pas à tous les motifs…
Figure 19.2 pour un exemple sur un autre motif

idee : utiliser un tableau next[M] qui indique de combien de caractères reculer en cas de mismatch
pour j>0, next[j] est le k maximum (k<j) pour lequel les k premiers caracteres du motif matchent les k derniers caracteres des j premiers caractères du motif.
et next[0] = -1

```
int kmpsearch(const char *p, const char *a)
{
  int i, j, M = strlen(p), N = strlen(a) ;
  initnext(p) ;
  for (i=0 , j=0 ; j<M && i<N ; i++,j++) {
    while ((j >= 0) && (a[i] != p[j])) {
      j = next[j];
    }
  }
  if (j == M)
    return i-M ;
  else
    return i ;
}
```

next[0]=-1 permet de rester sur le premier caractère du motif quand il ne matche pas.

initnext() ressemble a kmpsearch mais matche le pattern et lui-même :

```
void initnext(const char *p)
{
  int i, j, M = strlen(p) ;
  next[0] = -1 ;
  for (i=0 , j=-1 ; i<M; i++, j++, next[i]=j) {
    while ((j >= 0) && (p[i] != p[j])) {
      j = next[j];
    }
  }
}
```

Si le motif est connu à l'avance, on peut coder la recherche « en dur »

```
int kmpsearch(const char *a)
{
  int i = -1 ;
sm : i++ ;
s0 : if (a[i] != '1') goto sm ; i++ ;
s1 : if (a[i] != '0') goto s0 ; i++ ;
s2 : if (a[i] != '1') goto s0 ; i++ ;
s3 : if (a[i] != '0') goto s1 ; i++ ;
s4 : if (a[i] != '0') goto s2 ; i++ ;
s5 : if (a[i] != '1') goto s0 ; i++ ;
s6 : if (a[i] != '1') goto s1 ; i++ ;
s7 : if (a[i] != '1') goto s1 ; i++ ;
  return i-8 ;
}
```

Exercice : modifier initnext pour qu'il écrive la fonction kmpsearch… on fait un programme qui génère un programme !

On peut voir ce programme comme une machine à états finis : Figure 19.3

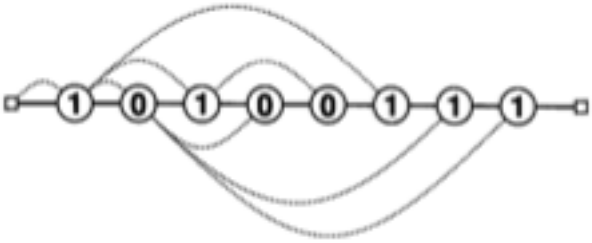


Figure 19.3 Finite state machine for the Knuth-Morris-Pratt algorithm.

Les états sont les cercles, les transitions les lignes. En trait plein : transition « matche ». En pointillés, transition « ne matche pas »
Amélioration : on a oublié d'utiliser le fait qu'on connaît le caractère qui a causé « ne matche pas » : si on cherche 1011, on a un mismatch sur le dernier 1, mais inutile de comparer à nouveau ce dernier caractère.
Nouvelle machine à états figure 19.4

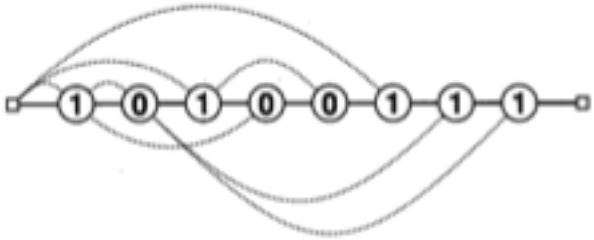


Figure 19.4 Knuth-Morris-Pratt finite-state machine (improved).

il suffit de remplacer dans initnext() l'instruction next[i] = j par next[i] = (p[i] == p[j]) ? next[j] : j ;
Complexité : jamais plus de M+N comparaisons, car on incrémente j ou on le remplace par next[j] au plus une fois pour chaque i
Mais l'algo KMP n'est pas tres efficace pour les motifs ne comportant pas beaucoup de repetitions (à quoi ressemble la machine à états ?), par contre il a l'avantage d'être séquentiel : on ne retourne jamais en arrière dans la chaîne passée en entrée

Algorithme de Boyer-Moore

On matche tous les caractères du motif de droite à gauche, et si on a un mismatch on peut directement avancer de next[j] caracteres. exemple avec 10100111, si on matche les trois derniers, on peut avancer de 7 : next[2] = 7
Autre exemple ;, sur une vraie chaîne, Figure 19.7

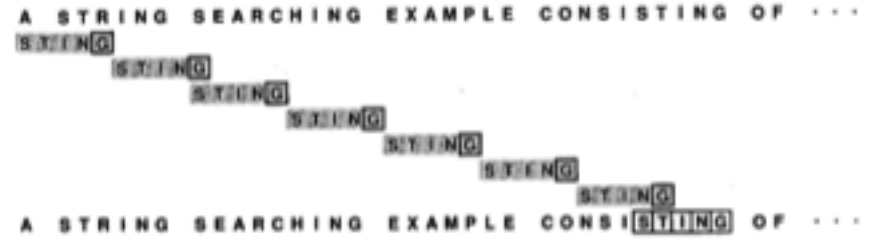


Figure 19.7 Boyer-Moore string search using the mismatched character heuristic.

Le tableau skip dit pour chaque lettre de l'alphabet de combien avancer si ce caractère apparaît dans le texte.
index(char c) returne 0 pour les blancs et i pour la i-ème lettre de l'alphabet.

initskip() initialise skip[] à M pour les caracteres qui ne sont pas dans le pattern, et, pour j de 0 à M-1, skip[index(p[j])] est initialisé à M-j-1

```
int mischarsearch(const char *p, const char *a)
{
  int i, j, M = strlen(p), N = strlen(a) ;
  initskip(p) ;
  for (i=M-1 , j=M-1 ; j>0; i--,j--) {
    while (a[i] != p[j]) {
      t = skip[index(a[i])] ;
      i += (M-j > t) ? M-j : t ;
      if (i >= N) return N ;
      j = M-1 ;
    }
  }
  return i ;
}
pour STING :
skip[G]=0
skip[N]=1
skip[I]=2
skip[T]=3
skip{S]=4
skip[autre]=5
```

Complexité : jamais plus de M+N comparaisons, et N/M étapes si l'alphabet n'est pas trop petit et le motif pas trop long.
Pas très utile pour les chaînes binaires.
et pour l'ADN ? quel est l'alphabet ?

Solution des exercices

Exercice 1.1 : maximum de 3 nombres sous forme de fonction directement : algorithme efficace ?

Discussion : pas de besoin de faire un tri, seuls 3 cas possibles, donc 2 test (2^2>3) suffisent
1. Input n1, n2, n3
2. If n1<=n2
   then max = n2
   else max = n1
3. If max < n3
   then max = n3.
4. Output max

Exercice 1.2 : Ecrire l'algorithme de calcul de F(n) efficace (en utilisant une boucle, pas de récursivité)
En Ada :

```
function F(N : Integer) return Integer ;
function F(N : Integer) return Integer is
      A : Integer := 0 ;
      B : Integer := 1 ;
      C : Integer;
   begin
      if N = 0 then
            return 0 ;
      elsif N = 1 then
            return 1 ;
      end if ;
      for I in 2..N loop
```

```
            C := A+B;
            A := B;
            B := C;
       end loop ;
       return C;
    end F ;
```

Exercice 3.1 : Ecrire la procédure qui imprime toutes les combinaisons des lettres d'une chaîne.

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Test_Combinaisons is

procedure Imprime (Les_Car : String; Racine : String := "") is
   --  imprime toutes les chaînes prolongeant Racine avec les différentes
   --  combinaisons des caractères de Les_Car

   Nombre_A_Placer : Positive := Les_Car'Last; -- nbre de car qui
                                               --   manquent  encore  à  Racine  pour
imprimer

begin
   if Les_Car'Length = 0 then
      Put_Line (Racine) ;
   else
      for I in Les_Car'Range loop
         Imprime  (Les_Car(Les_Car'First..I-1)&Les_Car(I+1..Nombre_A_Placer),  Racine
& Les_Car(I));
      end loop ;
   end if ;
end Imprime ;

begin
   Put_Line("combinaisons:");
   Imprime("ABC");
end;
```

Preuve par induction de la correction de cette procédure.

Exercice 4.1 : Insertion d'un code de recherche dichotomique dans un programme complet fourni par ailleurs.

```
procedure Dicho
   (Tt :    Vecteur;
    Xx :    Integer;
    Ind : out Boolean;
    Pos : out Indice) is
    Pstart : Indice := Tt'First;
    Pend : Indice:= Tt'Last;
    Pm : Indice;
  begin
    Ind := False;
    while Pstart < Pend loop
      Pm := (Pend + Pstart)/2;
      if Tt(Pm) < Xx then
        Pstart := Pm + 1;
      else
        Pend := Pm;
      end if;
      if Tt(Pstart) = Tt(Pend) and then Tt(Pm) = Xx then
        Ind:= True;
        Pos:=Pstart;
      end if;
```

```
    end loop;

  end Dicho;
```

Exercice 5.1 : Ecriture en Ada d'un paquetage de lecture avec reprise en cas d'erreur de saisie (lire numéro de code trois fois max, et invalidation si non numérique).

```
package Lecture_Sure is
   function Lire_Code (invite : string; Nb_Max: Integer := 3) return
Integer;
   -- cette fonction lit sur le support d'entrée standard un nombre avec
nb_max tentative;
   --       si échec au bout de ces tentative, le retour est le plus petit
entier possible (négatif)

end Lecture_Sure ;

============== attention, ceci ci dessous est dans un autre fichier compile
séparément

with Ada.Text_IO;
use Ada.Text_Io;
with ada.integer_text_IO; use ada.Integer_Text_IO;

package body Lecture_Sure is
   function Lire_Code (Invite : String; Nb_Max: Integer := 3) return
Integer is

      Valeur : Integer;
      Texte : String (1..20);
      lg_texte : natural;
   begin
      Put_Line (Invite);
      for I in 1..Nb_Max loop
         -- tant que le nombre n'est pas lu correctement, on réessaie
         begin
            -- alternative : Get_line (texte, lg_texte); get
(texte(1..lg_texte), valeur, lg_texte);
        Get (Valeur) ;
            return valeur; -- ici la lecture c'est bien passée : fin
         exception
             when Data_Error =>
            Skip_Line ; -- on purge la ligne erronée
      Put_Line ("Erreur, attention, entrer une valeur numérique");
            Put_Line (Invite);
         end;
      end loop;
      -- A ce stade, on abandonne
      Put_Line ("désolé, veuillez consulter votre conseiller");
      return Integer'First;
   end Lire_Code;

   end Lecture_Sure;
```

-- programme de test :

```
 with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
with  lecture_sure; use lecture_sure;

procedure Test_Lecture is
```

```
   X : Integer;

begin
   X := Lire_Code("merci de donner votre numéro secret");
   Put_Line ("le code lu est "& Integer'Image (X));
end Test_Lecture;
```

Exercice 6.1 : Fusion de deux tableaux ou de deux fichiers

1- Construire l'algorithme de fusion de deux tableaux (d'entiers) triés
(cf. corps de la procédure fusion du corrigé)

2- Comment généraliser à la fusion de deux entités (tableaux, fichiers, listes…) pour lesquelles seule une opération d'entrée (lire la valeur de l'élément courant - élément du tableau, ligne de fichier… - et passer à l'élément suivant) et une opération de sortie (écrire élément dans tableau ou dans fichier) sont possibles. Quel est l'intérêt de cette solution ?
permet de fusionner tous types d'objets, pas seulement des tableaux mais aussi des listes, des fichiers…).

3- On souhaite écrire un paquetage générique permettant de fusionner toutes les entités : quels doivent être les paramètres génériques ?
type de l'entité (fichier, tableau d'entier, ...)
type d'élément
fonction d'ordre (une seule suffit, par exemple « > »)
fonction de lecture d'un élément dans l'entite (entrée)
fonction d'écriture d'un élément dans l'entité (sortie)

Exercice 8.1 : Ecrire l'itération de boucle pour la renverse de liste sur place
temp := fin.suiv;
fin.suiv := debut;
debut := fin;
fin := temp;

Exercice 8.2 : on a une fonction (voir Ada.Numerics.Discrete_Random)

```
function Aleat (K : Natural) return Natural ;
--  retourne un nombre au hasard entre 0 et K, bornes comprises.
```

On veut imprimer une permutation aléatoire des nombres de 0 à N. Comment représenter E, l'ensemble des nombres qu'il reste à imprimer ?
Réponse 1: on construit une liste chaînée des nombres de 0 à N. à chaque tirage, on retire l'élément pour le placer dans la liste de sortie et on décrémente K. Ca nécessite à chaque itération le parcours de K/2 éléments en moyenne
Réponse 2 : (plus efficace !) on construit un tableau contenant les nombres restant à tirer initialisation : de 0 à N), et à chaque itération on tire un nombre I entre 0 et K, on lit la case I du tableau, puis on remplace le contenu de la case I par celui de la case K. On décrémente K.

Exercice 9.1 : Un calcul de maximum.
La fonction suivante, écrite en Ada, calcule l'élément maximal d'un vecteur T . L'appel initial est ValMax(T,K) où K est l'indice du dernier élément de T :

```
   type Tableau is array(Integer range <>) of Float;

   function ValMax (T: Tableau, I: Integer) return Float is
   begin
```

```
      if I = T'First then
         return T(T'First);
      elsif T(I) > ValMax(T, I-1) then
         return T(I);
      else
         return ValMax(T, I-1);
      end if;
   end ValMax;
```

Un utilisateur, qui teste cette fonction sur des vecteurs de tailles croissantes, vous dit : « je ne comprends pas, la fonction marche jusqu'à 27 éléments, mais semble boucler pour 28 ».
Comment interprétez-vous ce comportement ?
Que faut-il faire pour y remédier ?

Développer l'arbre d'appels de fonctions de valmax pour quelques exemples
Mettre en équation la complexité de valmax dans le cas favorable (O(n) si le tableau est trié par ordre croissant) - cas défavorable O(2^n) si le tableau est trié par ordre décroissant.
Solution : utiliser une variable locale pour stocker valmax(i-1)

Exercice 9.2 : On désire trouver les deux éléments les plus grands d'un tableau T. On supposera que les données sont équidistribuées. Comparer la complexité des deux versions Algo1 et Algo2 ci-dessous réalisant ce calcul. Que peut-on en conclure ?

```
   type Tableau is array(Integer range <>) of Float;

   procedure Algo1(X : Tableau; Max1, Max2 : out Float) is
      Aux : Float;
   begin
      Max1 := X(X'Last);
      Max2 := X(X'Last - 1);
      if Max1 < Max2 then
        Aux := Max1 ;
        Max1 := Max2 ;
        Max2 := Aux ;
      end if ;
      for I in reverse X'First..(X'Last-2) loop
        if X(I) >= Max1 then
           Max2 := Max1;
           Max1 := X(I);
        elsif X(I) > Max2 then
           Max2 := X(I);
        end if;
      end loop;
   end Algo1;
```

La seconde version Algo2 consiste à remplacer la partie intérieure de la boucle par :

```
        if X(I) >= Max2 then
           if X(I) >= Max1 then
              Max2 := Max1;
              Max1 := X(I);
           else
              Max2 := X(I);
           end if;
        end if;

   procedure Algo2(X : Tableau; Max1, Max2 : out Float) is
      Aux : Float;
   begin
      Max1 := X(X'Last);
      Max2 := X(X'Last - 1);
      if Max1 < Max2 then
        Aux := Max1 ;
        Max1 := Max2 ;
        Max2 := Aux ;
      end if ;
      for I in reverse X'First..(X'Last-2) loop
```

```
      if X(I) >= Max2 then
         if X(I) >= Max1 then
            Max2 := Max1;
            Max1 := X(I);
         else
            Max2 := X(I);
         end if;
      end if;
    end loop;
end Algo2;
```

Nombre de cas possibles et complexité des algos ;
Complexité la pire : tableau trié en ordre décroissant, 2(n-2)+1=2n-3 comparaison
Complexité en moyenne, version 1 :
second test effectué quand x(i)<Max1, donc avec une probabilité (n-i)/(n-i+1) (probabilité pour que le plus grand de (n-i+1) éléments ne soit pas le premier), donc, après changement de variable j=n-i+1 :

$$C_{moy}(n) = 1 + \sum_{j=3}^{n}\left(1\frac{1}{j} + 2\frac{j-1}{j}\right) = 1 + n - 2 + \sum_{j=3}^{n}\frac{j-1}{j} = 2n - 2 - H_n + \frac{1}{2}$$

avec $H_n = \sum_{i=1}^{n}\frac{1}{i} \approx \log(n)$

Complexité en moyenne, version 2 :
second test effectué quand x(i) est soit le premier, soit le second maximum parmi les n-i+1 derniers éléments, donc avec une probabilité 2/(n-i+1):

$$C_{moy}(n) = 1 + n - 2 + \sum_{j=3}^{n}\frac{2}{j} = n - 4 + 2H_n$$

La version 2 est donc en moyenne plus rapide, même si la complexité est du même ordre (O(n)). Attention cependant, nous n'avons pas tenu compte des affectations !