Compléments d'algorithmique

L1 MPCIE

Partie 1 : types composés

Types composés : Tableaux (rappels)

Un **type composé** est un type dont les variables contiennent plusieurs valeurs.

Un **tableau** est un type composé dans lequel toutes les valeurs sont **du même type**. Chaque valeur est repérée par un ou plusieurs indices inférieurs aux dimensions spécifiées dans la déclaration des types.

Exemple de déclaration

```
const int LMAX = 100;
const int CMAX = 50;
using ligne = std::array <int,CMAX>;
using matrice = std::array <ligne,LMAX>;
```

Exemple d'accès

```
matrice M;
std::cin >> M[5][2];
std::cin >> M;
```

← correct

← incorrect

Types composés : Enregistrements

Un **enregistrement** (ou **structure**) est un type composé dans lequel les valeurs peuvent être de types différents. Chacune de ces valeurs est appelée **champ**. Chaque champ est repéré par un nom.

Un enregistrement permet de définir un type qui regroupe des données (éventuellement de types différents) qui forment un tout et que l'on désire manipuler comme *une* variable.

Exemple :

Informations sur un étudiant :

Numéro INE	: Entier
Nom	: Chaine de caractères
Prénom	: Chaine de caractères
Notes	: Tableau de réels

Types composés : Enregistrements

Syntaxe C++ :

```
struct <nomtype> {
<type_champ1> <nom_champ1>;
<type_champ2> <nom_champ2>;
...
<type_champn> <nom_champn>;
};
```

Exemple :

```
const int maxnotes = 20;
using tabnotes = std::array <float,maxnotes>;

struct etudiant {
  int ine;
  std::string nom, prenom;
  tabnotes notes;
  int nbnotes;
};
```

Accès aux valeurs :

On désigne le champ d'un enregistrement en utilisant le point :

<nom_variable>.<nom_champ>

Types composés : Enregistrements

Exemple

```
#include <iostream>
#include <array>
#include <string>

const int maxnotes = 20;
using tabnotes = std::array <float,maxnotes>;

struct etudiant {
  int ine;
  std::string nom, prenom;
  tabnotes notes;
  int nbnotes;
};


void saisie (etudiant& e)
{
       std::cout << "Num INE : ";
       std::cin >> e.ine;
       std::cout << "Nom : ";
       std::cin >> e.nom;
       std::cout << "Prenom : ";
       std::cin >> e.prenom;
       e.nbnotes = 0;
}
```

```
void ajoute_note (etudiant& e, float note)
{
       e.notes[e.nbnotes] = note;
       ++e.nbnotes;
}

int main ()
{
       etudiant e;
       float n;

       saisie (e);
       std::cout << "Nom : " << e.nom << std::endl;
       std::cout << "Note : ";
       std::cin >> n;
       ajoute_note(e,n);
       std::cout << "Premiere note de " << e.prenom << " «
                 << e.nom << " : " << e.notes[0] << std::endl;

       return 0;
}
```

Compléments d'algorithmique

L1 MPCIE

Partie 2 : récursivité

Récursivité

Calculer la factorielle d'un entier *n* donné :

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

Définition itérative :

$$n! = \prod_{i=1}^{n} i$$

Définition récursive :

$$0! = 1$$
$$n! = (n-1)! \times n \text{ (pour } n > 0)$$

Pour définir la fonction factorielle, on utilise la fonction factorielle

Exemple : factorielle de 4

4! = 3! × 4
4! = (2! × 3) × 4
4! = ((1! × 2) × 3) × 4
4! = (((0! × 1) × 2) × 3) × 4
4! = (((1 × 1) × 2) × 3) × 4
4! = 24

Récursivité

Un algorithme (ou un sous-programme) est dit **récursif** quand il contient un (ou plusieurs) appel(s) à lui-même.

La récursivité peut être **indirecte** dans le cas où un sous programme sp_1 fait appel à un sous programme sp_2 qui fait lui-même appel à sp_1.

Cas d'utilisation de la récursivité :

- On peut décomposer le problème en sous-problèmes de même nature et plus court à traiter.
- Dans la décomposition en sous-problèmes, on doit toujours arriver à un cas où on peut résoudre le problème sans faire un appel récursif. Ce cas est appelé cas d'arrêt.

Si un sous-programme fait dans tous les cas un appel récursif, l'exécution ne se termine jamais.

Exemple du calcul de la factorielle

Le calcul de la factorielle (n!) peut être décomposé en :

- une multiplication
- un problème de même nature et plus court à traiter : $(n - 1)$!

Au bout d'un certain nombre de décompositions, on arrive toujours à 0!, que l'on peut calculer sans faire de décomposition.

Récursivité en C++

Aucune syntaxe particulière n'est requise : un appel récursif est réalisé de la même manière qu'un appel traditionnel à un sous-programme.

Exemple (factorielle) :

```
int fact (int n)
{
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

Exécution de fact(4)

fact(4) ← expression à évaluer
= **4 * fact(4-1)**
= 4 * fact(**3**)
= 4 * (**3 * fact(3-1)**)
= 4 * (3 * fact(**2**))
= 4 * (3 * (**2 * fact(2-1)**))
= 4 * (3 * (2 * fact(**1**)))
= 4 * (3 * (2 * (**1 * fact(1-1)**)))
= 4 * (3 * (2 * (1 * fact(**0**))))
= 4 * (3 * (2 * (1 * **1**)))
= 4 * (3 * (2 * **1**))
= 4 * (3 * **2**)
= 4 * **6**
= **24** ← **résultat**

Exponentiation

Il est possible de calculer x^n en utilisant uniquement les propriétés suivantes :

$$x^0 = 1$$
$$x^n = x \times x^{n-1} \text{ (si } n > 0)$$

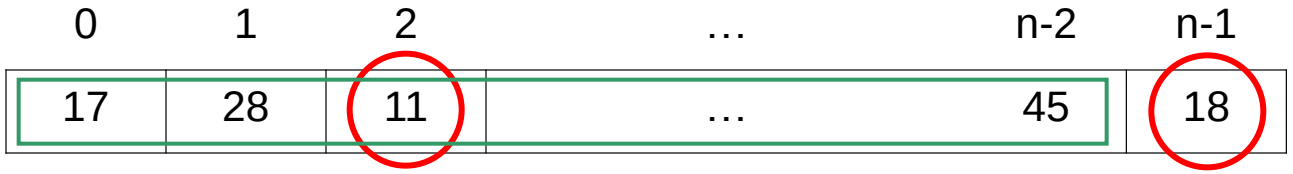
```
float puissance (float x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * puissance(x,n-1);
}
```

Recherche d'un minimum dans un tableau

Problème : Rechercher la valeur minimum dans un tableau d'entiers.

Comment l'écrire en récursif ?

Décomposer un problème en un problème plus simple :
Le minimum parmi n valeurs est le minimum entre :
- le minimum parmi les n-1 premières valeurs, et
- la $n^{ième}$ valeur



Si le minimum *a* du tableau privé d'un élément est inférieur à la valeur *b* de cet élément restant (indice n-1 : 18), alors le minimum du tableau est *a*, sinon c'est *b*.

Cas d'arrêt : Résolution triviale pour un tableau de taille 1
la valeur minimale parmi 1 valeur est cette unique valeur (pas d'appel récursif).

Recherche d'un minimum dans un tableau

```
const int nmax = 100;
using tab = std::array <int,nmax>;

int minimum (tab t, int n)
{
  if (n == 1)
    return t[0];
  else {
    int m = minimum(t,n-1);
    if (m < t[n-1])
      return m;
    else
      return t[n-1];
  }
}
```

Fonctions et procédures récursives

Une fonction récursive a souvent la forme suivante :

si *condition de cas d'arrêt*

alors retourner *calcul simple sans appel récursif*

sinon *calcul faisant intervenir le résultat d'un appel récursif*

retourner *résultat du calcul*

Un sous-programme récursif n'est pas nécessairement une fonction qui calcule et retourne une valeur. Dans le cas d'une procédures récursives, seuls des traitements sont effectués, sans valeur de retour.

Algorithmes récursifs / algorithmes itératifs

Tout ce qui peut être écrit en récursif peut être écrit en itératif, et inversement.

Quand utiliser la récursivité ?

Quand la décomposition est simple à trouver et à comprendre.

Quand ne pas utiliser la récursivité ?

Quand il existe une solution itérative simple et intuitive.

Quand la profondeur de la récursivité est trop grande (dépassement possible de la *pile d'exécution*)

Exemple : recherche du minimum dans un tableau de 100 000 éléments :

Algorithme **récursif** : 100 000 appels.

Algorithme **itératif** : un simple parcours (boucle) des éléments.

La solution itérative est préférable.

Algorithmes récursifs / algorithmes itératifs

L'appel récursif doit toujours porter sur un problème plus court à traiter, qui se rapproche d'un cas d'arrêt.

Exemple – Utilisation de la propriété $x^n = x^{n+1}/x$ pour calculer une puissance.

```
float puissance (float x, int n)
{
  if (n == 0)
    return 1;
  else
    return puissance(x,n+1) / x;
}
```

Cette fonction **ne se termine jamais** pour un appel avec $n > 0$.

Somme des valeurs d'un tableau

Problème : Écrire (en récursif) un sous-programme qui calcule la somme des éléments d'un tableau de réels.

Appel récursif : L'opérateur + ne peut être utilisé qu'avec deux réels.
La somme des valeurs d'un tableau de *n* éléments est égale à la somme des valeurs des *n*-1 premiers éléments ajoutée à la valeur du $n^{ième}$ élément.

Cas d'arrêt : Dans un tableau de taille 1, la somme des éléments est égale à la valeur de l'unique élément.

Version récursive

```
float somme (tab t, int n)
{
  if (n == 1)
    return t[0];
  else
    return somme(t,n-1) + t[n-1];
}
```

Version itérative

```
float somme (tab t, int n)
{
  float s = 0;
  for (int i = 0; i < n; ++i)
    s = s + t[i];
  return s;
}
```

Moyenne des valeurs d'un tableau

En itératif, pour calculer la moyenne des éléments après le calcul de la somme des éléments, il suffit de diviser la somme par le nombre d'éléments.

Comment procéder dans un programme récursif ?

Exemple incorrect :

```
float moyenne (tab t, int n)
{
  if (n == 1)
    return t[0];
  else
    return (moyenne(t,n-1) + t[n-1])/n;
}
```

Cette fonction est incorrecte car la division par *n* est réalisée à chaque appel récursif et donc avec une valeur de *n* différente.

Exemple correct :

```
float moyenne (tab t, int n)
{
  if (n == 1)
    return t[0];
  else
    return ((n-1) * moyenne(t,n-1) + t[n-1])/n;
}
```

Recherche dichotomique

Rechercher une valeur dans un tableau trié.

Problème : Déterminer si une valeur entière donnée apparait parmi les éléments d'un tableau dont les éléments sont triés par ordre croissant.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
5	8	11	17	24	28	46	53	58	62

Recherche séquentielle : Parcourir tous les éléments du tableau jusqu'à ce que l'on trouve la valeur cherchée.

Inconvénient : si le tableau est très grand, la recherche peut prendre du temps.

Le tableau étant trié, on peut écrire une fonction de recherche plus efficace, qui considère *plus d'une case* à chaque étape.

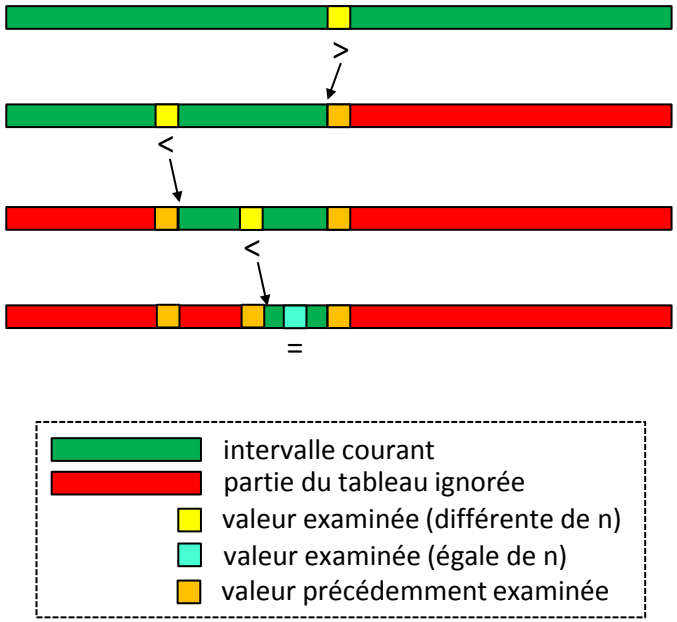
Recherche dichotomique : l'ensemble de valeurs sur lequel porte la recherche est coupé en deux à chaque étape, la recherche continuant sur *un seul* de ces sous-ensembles.

Recherche dichotomique

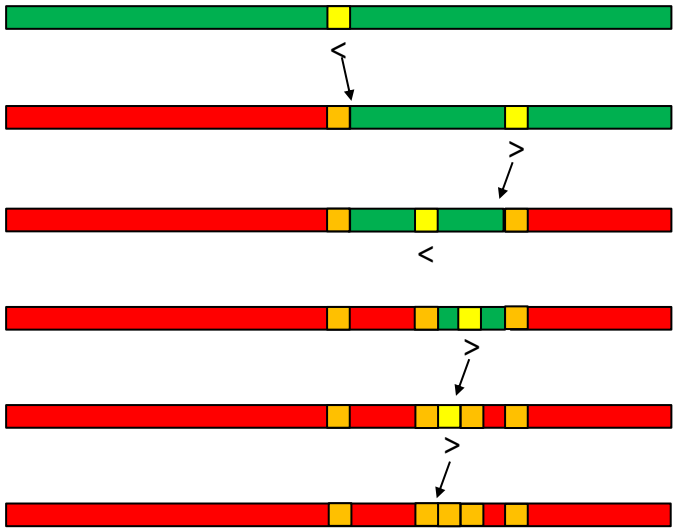
Principe général

- On recherche n dans l'intervalle d'indices [debut,fin].
- On détermine l'indice médian (au milieu) : milieu = (debut + fin) / 2 [division entière].
- On compare la valeur de l'élément central (à l'indice t[milieu]) à n.
    - S'il s'agit de l'élément recherché, la recherche s'arrête avec succès (élément trouvé).
    - S'il l'élément central t[milieu] est supérieur à n, on poursuit la recherche sur la moitié inférieure de l'intervalle : [debut , milieu – 1] ; sinon, sur la moitié supérieure [milieu + 1 , fin]…
    - … à moins que l'intervalle en question ne contienne aucune valeur ; dans ce cas la recherche s'arrête (élément absent).

Cas 1 : succès



Cas 2 : échec



Recherche dichotomique

Exemple : déterminer si **28** appartient au tableau t.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
t	5	8	11	17	24	28	46	53	58	62

- **Intervalle : [0, 9]**
  Milieu : 4 ; t[4] = 24
  24 est strictement inférieur à la valeur recherchée 28.
  Comme le tableau est trié dans l'ordre croissant, aucune valeur comprise entre t[0] et t[3] ne peut être supérieure à 24, ni par conséquent égale à 28.
  On continue donc la recherche dans l'intervalle d'indices [5,9].

- **Intervalle : [5,9]**
  Milieu : 7 ; t[7] = 53
  53 > 28, donc la recherche se poursuit dans l'intervalle [5,6].

- **Intervalle : [5,6]**
  Milieu : 5 ; t[5] = 28
  La valeur cherchée est trouvée.
  Si, par exemple, t[5] avait été strictement supérieur à 28, la recherche aurait pu être stoppée (l'intervalle à suivre est vide) avec comme résultat que la valeur cherchée n'appartient pas à t.

Recherche dichotomique

```
bool recherche (int v, tab t, int d, int f)
// renvoie vrai ssi v apparait dans t entre les indices d et f
// on suppose les valeurs de t triées par ordre croissant
{
  if (d > f)
     return false;
  else {
     int m = (d + f) / 2;
     if (t[m] == v)
        return true;
     else if (t[m] > v)
        return recherche(v,t,d,m-1);
     else return recherche(v,t,m+1,f);
  }
}
```

→ **Analyser le déroulement des exécutions de recherche(11,t,0,9) et recherche(65,t,0,9)**

	0	1	2	3	4	5	6	7	8	9
t	5	8	11	17	24	28	46	53	58	62

Recherche dichotomique

La recherche séquentielle d'une valeur dans un tableau de $\boldsymbol{n}$ éléments s'effectue en $\boldsymbol{n}$ **étapes au plus**, tandis que la recherche dichotomique s'effectue en $\boldsymbol{\lfloor \log_2 n \rfloor + 1}$ **étapes au plus**.

Nombre d'éléments (taille du tableau)	Recherche séquentielle (n)	Recherche Dichotomique ($\lfloor \log_2(n) \rfloor + 1$)
10	10	4
1 000	1 000	10
100 000	100 000	17

Sur des petits tableaux, une recherche par dichotomie n'est pas ou guère plus rapide (jusqu'à 10 étapes contre 4 pour *n* = 10, mais quelques calculs supplémentaires pour déterminer les intervalles).

Sur des grands tableaux, la dichotomie est largement plus rapide (jusqu'à 100 000 étapes contre 17 pour *n* = 100 000).