Chapitre 5 : conception de programmes

Décomposition par raffinements successifs

Décomposition itérative

Décomposition récursive

Décomposition par raffinements successifs

L'analyse d'un problème procède par décompositions successives en sous-problèmes, eux-mêmes décomposés jusqu'à un niveau de problème dont la solution est triviale.

En programmation, à chaque sous-problème correspond un sous-programme.

Une telle décomposition décrit l'algorithme de résolution du problème

Décomposition par raffinements successifs : exemple (1/5)

Affichage des tables de multiplication.

Résultat attendu :

```
    Tables de multiplication
------------------------------
   |   1  2  3  4  5  6  7  8  9
1  |   1  2  3  4  5  6  7  8  9
2  |   2  4  6  8 10 12 14 16 18
3  |   3  6  9 12 15 18 21 24 27
4  |   4  8 12 16 20 24 28 32 36
5  |   5 10 15 20 25 30 35 40 45
6  |   6 12 18 24 30 36 42 48 54
7  |   7 14 21 28 35 42 49 56 63
8  |   8 16 24 32 40 48 56 64 72
9  |   9 18 27 36 45 54 63 72 81
```

Décomposition par raffinements successifs : exemple (2/5)

Algorithme :

raffinement 1er niveau :

```
- afficher le titre;
- afficher la ligne de tirets;
- afficher les numéros de table de 1 à 9;
- afficher le corps des tables;
```

Décomposition par raffinements successifs : exemple (3/5)

Algorithme :

raffinement 2nd niveau : `afficher le corps des tables`

```
- pour chaque chiffre de 1 à 9 répéter
    - afficher ce chiffre suivi de "|"
    - afficher une ligne de la table;
  fin pour;
```

Décomposition par raffinements successifs : exemple (4/5)

Algorithme :

raffinement 3ème niveau : `afficher une ligne de la table;`

```
- pour chaque chiffre de 1 à 9 répéter
      afficher valeur de la table correspondante;
  fin pour;
```

Décomposition par raffinements successifs : exemple (5/5)

Algorithme :

raffinement 4ème niveau : `afficher valeur de la table correspondante`

```
- si le produit des 2 chiffres<10
            afficher "   " suivi de la valeur i*j;
  sinon
            afficher " " suivi de la valeur i*j;
  fin si;
```

i : n° de ligne
j : n° de colonne

Décomposition par raffinements successifs : exemple (3/3)

```
import static java.lang.System.*;
public class Tables{
    public static void main( String[] args ){
        out.println("      Tables de multiplication  ");
        out.println("-------------------------------");
        out.print("  | ");
        for(int i=1;i<=9;i++) // les numéros de table de 1 à 9;
           out.print("  "+i);
        out.println();
        for(int i=1;i<=9;i++){ // affichage d'une ligne
          out.print(i+" | ");
          for(int j=1;j<=9;j++) affichage des valeurs de la ligne
            if(i*j<10)   out.print("  "+i*j);
            else         out.print(" "+(i*j));
          out.println();
        }
    }
}
```

Décomposition itérative

Il est très courant de partir d'une spécification de problème sous forme de propriétés de la fonction à calculer.

L'outil mathématique le plus précieux pour construire l'algorithme est, dans ce cas, le raisonnement par récurrence.

L'ensemble des propriétés que la fonction doit satisfaire est traduit par une *hypothèse de récurrence*.

On construit ainsi les programmes de manière raisonnée et systématique.

On dit que *P* est une propriété récurrente sur *N* si, étant vérifiée pour un entier *a*, elle est vérifiée pour tout entier supérieur ou égal à *a*.

Notion de récurrence

La construction d'algorithme par récurrence, où on procède pas à pas, chaque étape utilisant les résultats de la précédente, repose sur l'induction mathématique.

Soit P(n) une propriété sur les entiers naturels n N.
Si on peut montrer que :

H1 : P(a) est vraie pour un entier a.
H2 : la validité de P(n) entraîne celle de P(n+1)

alors P(n) est vraie pour tout entier naturel supérieur ou égal à a.

Méthode par récurrence

Première étape

- Déterminer une *hypothèse de récurrence*.

Elle formalise l'idée choisie pour construire la solution du problème donné.

Seconde étape

- Dégager le sous-ensemble ordonné de *N* à l'intérieur duquel *P* doit être vérifiée.

Cela revient à établir la *condition d'arrêt de l'itération*.

Troisième étape

- Démontrer que si *P (i)* est vérifiée (*i* appartenant au sous-ensemble ordonné), alors *P (suc(i))* est aussi vérifiée.

Cette troisième étape revient à construire le *corps de boucle*.

On procède en fait ainsi:

1- se rapprocher de la solution

2- retrouver l'hypothèse de récurrence.

Quatrième étape

- Le raisonnement se termine par la preuve que *P(1)* est vérifiée.

Cette quatrième étape correspond à l'*initialisation*.

Méthode par récurrence: résumé

démarche mathématique	*démarche informatique*
dégager P (ensemble des propriétés que la fonction doit satisfaire)	poser une hypothèse de récurrence : P
dégager le sous-ensemble ordonné S de N à l'intérieur duquel P doit être vérifiée	établir la condition d'arrêt
démontrer que si $P(i)$ et $i \in S$ est vérifiée alors $P(suc(i))$ est vérifiée	construction du corps de boucle: 1-se rapprocher de la solution 2-retrouver l'hypothèse de récurrence
monter que P(1) est vérifiée	initialisation

Invariant

En cas de succès du raisonnement, l'hypothèse de récurrence devient *l'invariant* de la boucle.

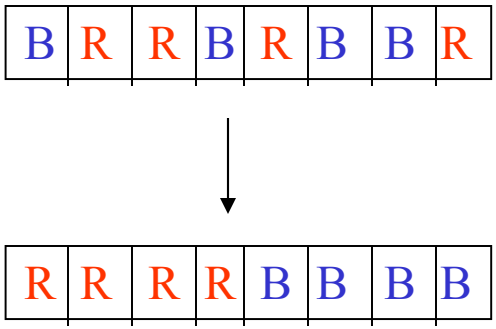
Cet invariant se transforme en un commentaire informatif du programme.

Les preuves de correction d'un programme sont facilitées par sa présence.

Exemple : tri

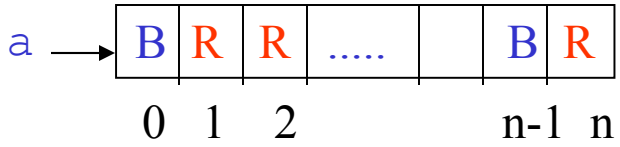
Un tableau de `n` éléments contient un certain nombre (éventuellement nul) de boules rouges et un certain nombre (éventuellement nul) de boules bleues.

On veut réordonner ce tableau de manière à ce que toutes les boules rouges soient à gauche du tableau et les blanches à droite



Exemple : hypothèse de récurrence

Soit `a`, noté `a[0..n]`, le tableau de boules rouges et bleues



La solution devra satisfaire la propriété :

```
( b [0..n]) [( i [0..b] a[i]=R)    ( i [b+1..n] a[i]=B) ]
```

1 - Choix de l'hypothèse de récurrence `H` :

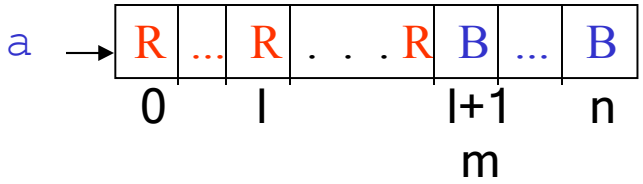
```
(  i [0..l]       a[i]=R
   i [l+1..m-1]   a[i]=R     a[i]=B
   i [m..n]       a[i]=B
)
```

Exemple : condition d'arrêt

2- Condition d'arrêt de l'itération :

$$H \quad l+1=m \Rightarrow \begin{pmatrix} i \in [0..l] & a[i]=R \\ i \in [l+1..n] & a[i]=B \end{pmatrix}$$

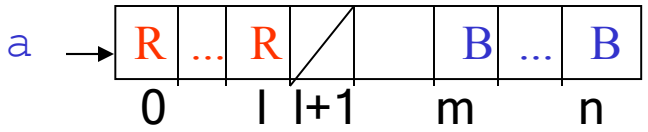
la solution doit vérifier cette propriété, d'où la condition d'arrêt : `l+1=m`



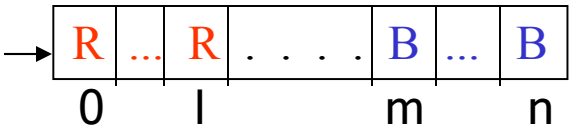
Exemple : corps de boucle(1/2)

A l'entrée de la boucle, la propriété suivante est vérifiée : H `l+1 m`

- placer l'élément `a[l+1]` rapproche de la solution



- reconstruire l'hypothèse de récurrence qui deviendra un invariant



Exemple : corps de boucle(2/2)

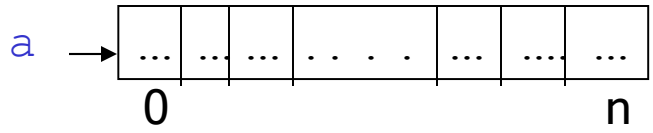
placer l'élément `a[l+1]:`

on sait que : `[ H  a[l+1]=R  a[l+1]=B ]`

```
si a[l+1]=R alors
        ne rien faire;
        retrouver l'invariant
        l←l+1;
        [H]
sinon
        permuter a[l+1] avec a[m-1]);
        retrouver l'invariant
        m←m-1;
        [H]
fin si
```

Exemple : initialisation

Au début les sous-tableaux ordonnés sont vides :



initialement :

```
l←-1
m←n+1
```

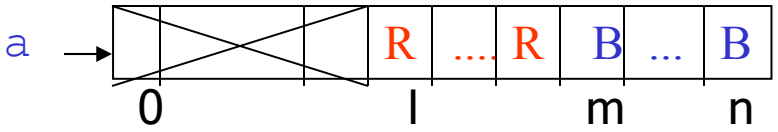
Exemple : programme (1/2)

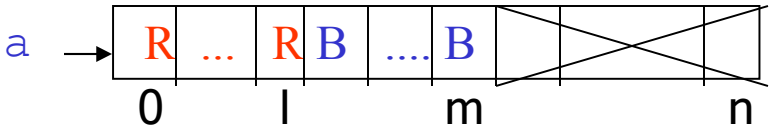
```
public class Drapeau{
  public static void main(String[] args){
    char[] t = {'B','R','R','B','B','R','B','R'};
    // afficher le tableau : voir transparent suivant
    // réordonner le tableau
    int l=-1;int m=t.length;
    //  H
    while (l+1 < m){
      // placer l'élément a[l+1]
      // [ H  a[l+1]=R   a[l+1]=B ]
      if (t[l+1]=='R')
        l=l+1;
        // H
      else{
        char tmp = t[l+1];t[l+1] = t[m-1];t[m-1] = tmp;
        m=m-1;
       // H
      }
    }
   // afficher le tableau : voir transparent suivant
  }
```

Exemple : programme (2/2)

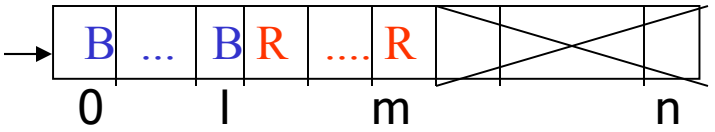
```
// afficher le tableau t
    System.out.print("[ ");
    for(int i=0;i<t.length;i++)
      System.out.print(t[i]+" ");
    System.out.println("]");
```

Autres hypothèses





mauvaise hypothèse a



Décomposition récursive

Notion de récursivité

Une décomposition est dite récursive si le sous-problème engendré est identique au problème initial mais pour un cas plus simple.
La solution du problème s'exprime par rapport à elle-même.

Exemple classique : le calcul de la factorielle d'un entier

`n!=1*2*3*....*(n-1)*n`

or

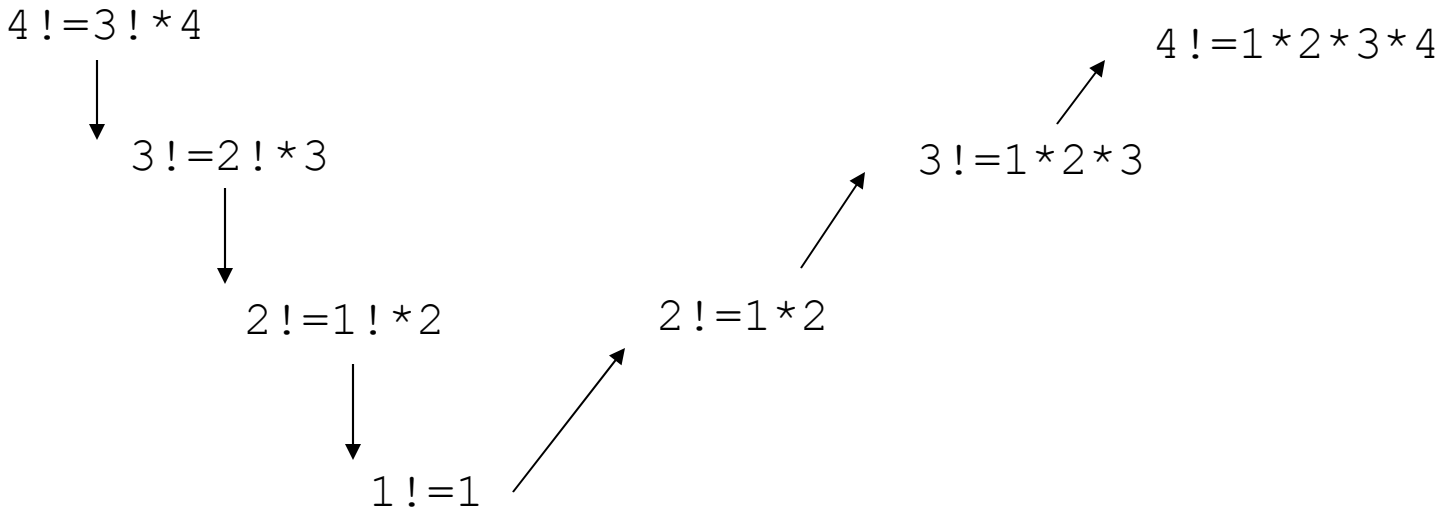
`(n-1)! = 1*2*3*....*(n-1)` cas plus simple car `(n-1)<n`

donc

`n!=(n-1)!*n` pour `n>0`

Décomposition récursive du calcul

Décomposition du calcul de factorielle 4 :



Toute décomposition récursive nécessite une solution triviale au niveau d'un sous-problème sinon la décomposition devient infinie
Le sous-problème étant résolu directement, le sous-problème du niveau supérieur l'est aussi et ainsi de suite

Décomposition récursive: méthode

- Trouver une décomposition récursive du problème
  - trouver l'élément de récursivité qui permet de définir les cas plus simples(une valeur numérique qui décroît, la taille de données qui diminue).
  - exprimer la solution en fonction de la solution pour le cas plus simple.
- Déterminer la condition d'arrêt de la récursivité
  - exprimer la solution dans ce cas
  - vérifier que la condition d'arrêt est atteinte après un nombre fini d'appels récursifs dans tous les cas
- Réunir les 2 étapes précédentes

Exemple 1

Calcul du nombre d'occurrences `n` d'un caractère donné `c` dans une chaîne de caractères `s`.

- décomposition récursive

  - élément de récursivité : la chaîne dont la taille diminue
  - un cas plus simple : la chaîne privée de son premier élément (notée `s1`)
  - solution en fonction du cas plus simple :

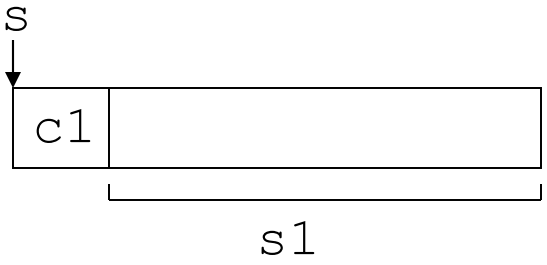
    `n1` est le nombre d'occurrences de `c` dans `s1`

    soit `c1` le premier caractère de `s` :

    `si` `c1=c` **`alors`** `n=n1+1` **`sinon`** `n=n1`

- condition d'arrêt : `s=""` (chaîne vide), `n=0` dans ce cas
- réunion des 2 étapes

```
nbOcc(c,s)= si vide(s) alors n=0
            sinon
              si c=c1 alors n=nbOcc(c,s1)+1
              sinon n= nbOcc(c,s1)
```



Exemple 2

Affichage des éléments d'un tableau dans l'ordre inverse

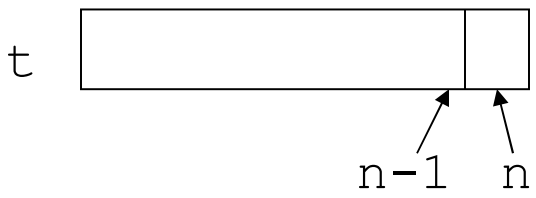
- décomposition récursive
  - élément de récursivité : l'indice du dernier élément `n` du tableau
  - un cas plus simple : le tableau jusqu'à l'indice `n-1`
  - solution en fonction du cas plus simple :

    `afficherInverse(t[0..n-1])`

    d'où la solution générale:

    `afficher(t[n]);afficherInverse(t[0..n-1]);`
- condition d'arrêt : `n=0`
- réunion des 2 étapes

```
afficherInverse(t[0..n])=
                    si n=0 alors afficher(t[0]);
                    sinon
                      afficher(t[n]);
                      afficherInverse(t[0..n-1]);
```



Itération vs récursion

Tout programme récursif peut-être écrit par itération.
Tout programme itératif peut-être écrit récursivement.

Itération :

- efficacité car le code compilé est proche du code machine correspondant
- raisonnement lié au modèle de fonctionnement de la machine : programmation impérative ou procédurale (décrire le "comment")

Récursivité :

- code produit généralement moins efficace
- raisonnement non lié à la machine (plus haut niveau d'abstraction) : programmation déclarative (décrire le "quoi")