UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Programmation avancée et Complexité

Frédéric Flouvat

Université de la Nouvelle-Calédonie

Présentation de l'EC

Objectifs : Approfondir les acquis en algorithmique/programmation et aborder la question de l'efficacité des algorithmes

Trois chapitres :

1. Rappels d'algorithmique
2. De Python au langage C
3. Complexité et ordre de grandeur

Volume horaire : 12h CM (6 séances) / 12h TD (6 séances) / 24h TP (12 séances)

- CM et TD mélangés (en fonction de la progression du cours)

Evaluation en Programmation Avancée :

- QCM en début de chaque cours (moyenne, coef. 0.1), un contrôle "papier" à la fin du cours (coef. 0.5), trois TP notés (moyenne, coef. 0.4)
- 2eme chance : un contrôle "papier" la dernière semaine de cours qui remplace les notes précédentes si la note est meilleure

Quelques références bibliographiques

T.H. Cormen.
Introduction à l'algorithmique : cours et exercices.
Sciences sup. Dunod, 2002.

Donald E. Knuth.
Art of Computer Programming (3rd Edition).
Addison-Wesley Professional, November 1997.

Un grand nombre de ressources sur internet

- Algorithmes récursifs, S. Vegel et M.-E. Voge, Université de Nice, 2007.
- C for Python programmers, C. Burch et E. Patitsas, Hendrix College et University of Toronto, 2012.
- ...

Plan

Programmation : le processus **classique**

1. Etude préalable : compréhension et modélisation du problème
2. Spécifier les données du problème et les résultats
   - décrire les informations en entrée (les données à utiliser/traiter) et en sortie (le résultat cherché)
3. Choisir une méthode pour résoudre le problème
   - 3.1 trouver des solutions/méthodes en langage naturel (pas informatique)
   - 3.2 décomposer en plusieurs algorithmes/modules
   - 3.3 écrire les algorithmes et décrire la représentation des données (cad les structures de données)
   - 3.4 choisir la méthode en fonction de critéres tels que l'efficacité ou la simplicité
4. Programmer dans un langage informatique
   - traduire la solution sous forme de programme
5. Tester et évaluer le travail réalisé
   - fonctionne-t-il correctement ? répond-il aux besoins initiaux ?
6. Documenter le logiciel

Qu'est-ce qu'un algorithme ?

Algorithme ⇨ méthode permettant de résoudre un problème de calcul bien spécifié

- p.ex. trier une suite de nombres, analyser de l'ADN ...
- exprimé dans un langage entre le langage humain et les langages informatiques

Exemple de problème : calculer x^n

- entrée : un réel x $(\neq 0)$ et un entier n
- sortie : le réel x^n

Exemple de résolution :

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: res ← 1
2: Pour i de 1 à n faire
3:     res ← res . x
4: Fin Pour
5: Retourner res
```

Qu'est-ce qu'un algorithme ?

Problématiques de l'algorithmique :

- trouver une méthode de résolution (exacte ou approchée) du problème
- trouver une méthode efficace

Remarque : algorithmes vs programmes

- un algorithme est une méthode décrite dans un langage proche du langage naturel
  - ex : "Pour i de 1 à n faire"
- un programme est la réalisation (i.e. l'*implémentation*) d'un algorithme via un langage de programmation
  - ex : "for(int i = 1 ; i < =n ; i++) " (en langage C)

Méthodologie de conception d'un algorithme

Analyse descendante

- décomposer un problème complexe en sous problèmes et ces sous problèmes en d'autres sous problèmes jusqu'à obtenir des problèmes faciles à résoudre

Garder à l'esprit

- la modularité : un module résout un petit problème donné et doit être réutilisable
  - dans notre cas module = fonction/procédure
- l'efficacité : étudier la *complexité* de l'algorithme

Représenter les données

Un algorithme décrit une méthode qui effectue des traitements (opérations) sur des informations (données)

⇨ la représentation des données est donc un aspect important en algorithmique

- une des étapes principales dans la construction d'un programme

Objectifs :

- représentation appropriée des données
  - des algorithmes moins complexes et plus efficaces
- réutilisation des mêmes structures de données dans beaucoup d'algorithmes

Algorithmes itératifs/récursifs : définitions

Pour une méthode de résolution, plusieurs algorithmes possibles
➪ p.ex. Itératif vs Récursif

Itératif : Un algorithme est dit itératif s'il utilise uniquement des boucles et/ou des instructions conditionnelles pour résoudre le problème

Récursif : Un algorithme est dit récursif s'il s'appelle lui-même (directement ou indirectement) une ou plusieurs fois pour traiter des sous-problèmes similaires

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n'
1: res $\leftarrow$ 1
2: **Pour** i de 1 à n **faire**
3: res $\leftarrow$ res . x
4: **Fin Pour**
5: **Retourner** res

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n
1: **Si** n = 1 **Alors**
2: **Retourner** x
3: **Fin Si**
4: res $\leftarrow$ Puissance(x, n-1) . x
5: **Retourner** res

Les différents types de récursivité

Récursivité simple :

- fonction/procédure qui s'appelle elle-même une seule fois

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 0 Alors
2:     Retourner 1
3: Fin Si
4: res ← Puissance(x, n-1) . x
5: Retourner res
```

"Récursivité" croisée :

Fonction Pair(n)

Entrée: un entier n
Sortie: vrai ou faux selon que n soit pair ou non

```
1: Si n = 0 Alors
2:     Retourner vrai
3: Sinon
4:     Retourner Impair( n-1)
5: Fin Si
```

Fonction Impair(n)

Entrée: un entier n
Sortie: vrai ou faux selon que n soit impair ou non

```
1: Si n = 0 Alors
2:     Retourner faux
3: Sinon
4:     Retourner Pair( n-1)
5: Fin Si
```

Les différents types de récursivité

Récursivité multiple :

- fonction/procédure effectuant plusieurs appels récursifs

Procedure AffichageRec2(i, tabmots)

Entrée: un entier i représentant un indice du tableau, un tableau de chaînes de caractères $tabmots$

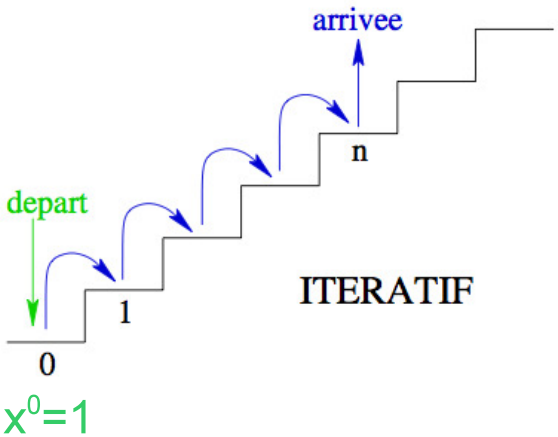
```
1: Si i < longueur[ tabmots ] Alors
2:     AffichageRec2( i+1, tabmots )
3:     AfficheEcran( tabmots[i] )
4:     AffichageRec2( i+1, tabmots )
5: Fin Si
```

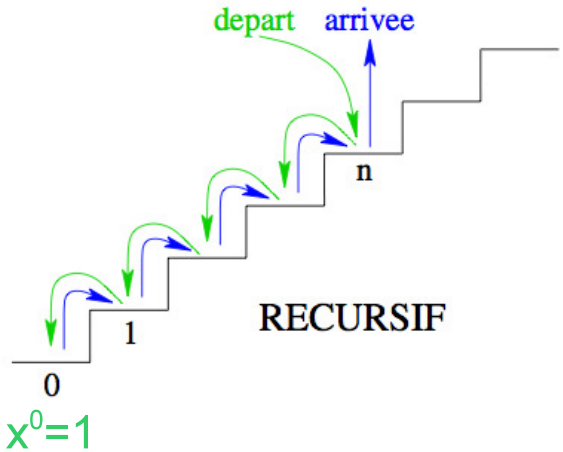
Exemple d'exécution avec ['un ','deux ', 'trois '] et $i = 0$:

⇨ affiche à l'écran : *trois deux trois un trois deux trois*

Principe de la récursivité simple : l'escalier

Exemple avec x^n (propriété $x^n = x^{n-1} \times x$)





- départ : information connue
- monte vers le résultat
- arrêt dés que le résultat est trouvé

⇨ boucle (pour, tant que, répéter, ...)

- départ : information cherchée
- descend vers l'information connue
- arrêt en bas
- monte vers le résultat
- arrêt dés que le résultat est trouvé

⇨ appels récursifs (algorithme monte/descend les marches)
⇨ mais comment ?

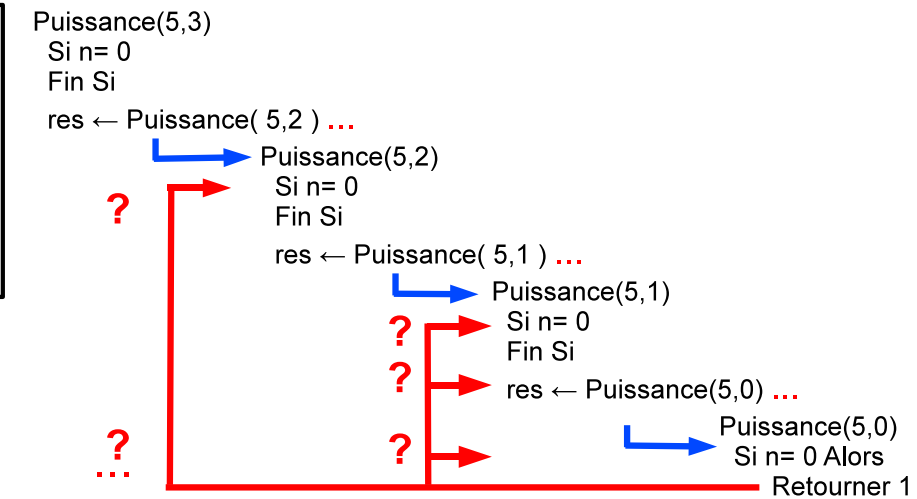
Fonctionnement de la récursivité simple

Utilisation d'une pile systéme (au niveau de la mémoire centrale) pour stocker les appels récursifs

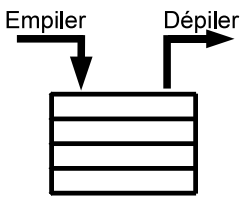
- permet au programme de se rappeler où il en était avant l'appel récursif

Fonction Puissance(x,n)

Entrée: un entier n≥ 0
Sortie: l'entier x^n
1: **Si** n = 0 **Alors**
2: **Retourner** 1
3: **Fin Si**
4: res ← Puissance(x, n-1) . x
5: **Retourner** res



Rappel : une pile est une structure de données de type tableau où la derniére information entrée est aussi la premiére à sortir



Fonctionnement de la récursivité simple

A chaque appel d'un algorithme récursif, les paramétres transmis par valeur et les variables locales sont empilés

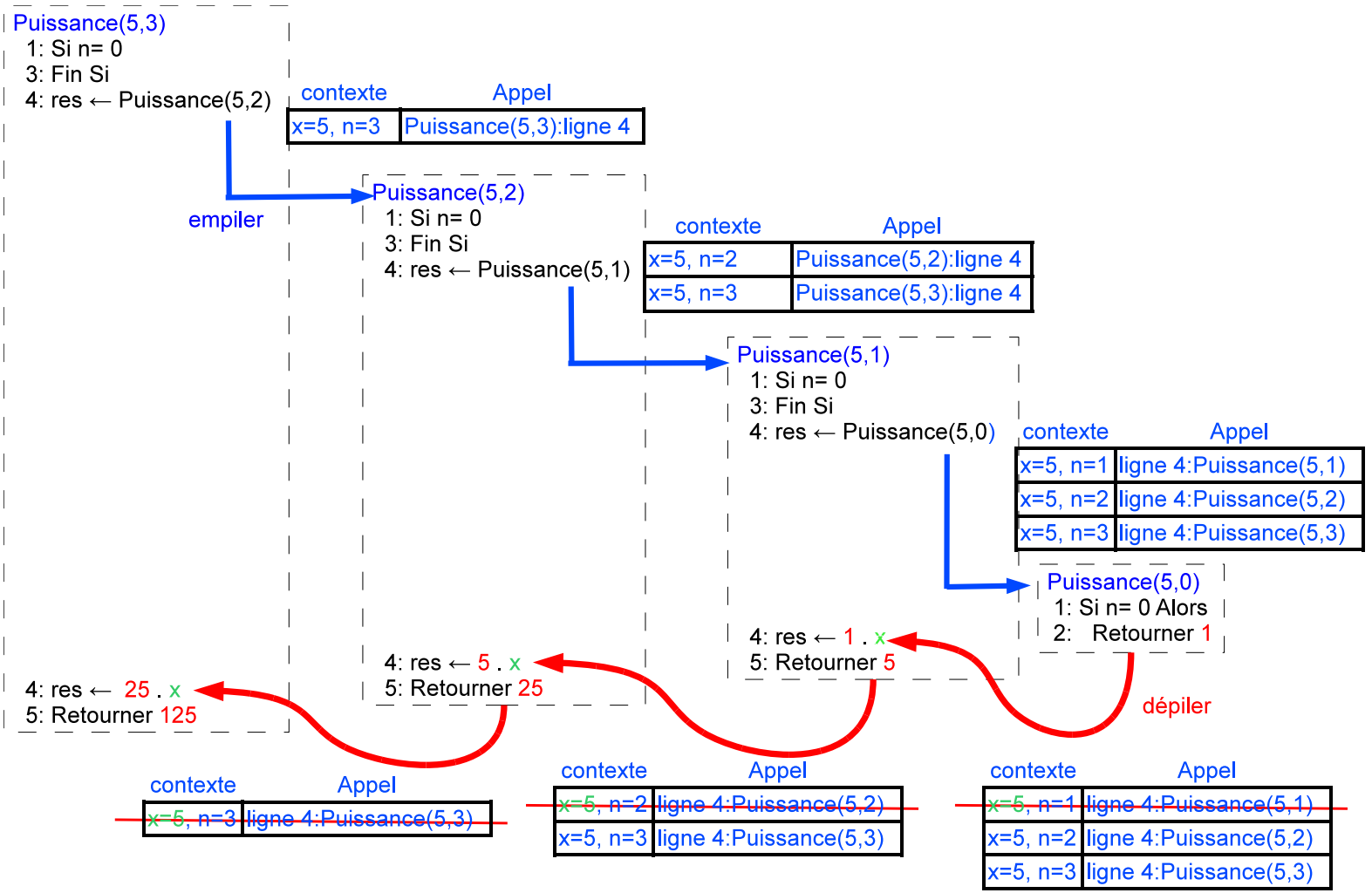
- paramétres + variables locales = contexte d'activation de l'appel

De manière plus générale, une exécution en 2 phases :

- empiler dans la pile jusqu'à arriver à un appel de la fonction pour lequel une condition d'arrêt est vérifiée
- dépiler en utilisant les résultats des appels récursifs pour terminer les appels précédents

Remarque : gestion de la pile transparente pour le programmeur et l'utilisateur

Exemple d'exécution avec une récursivité simple



Exercice

Ecrire la trace de l'exécution des deux algorithmes récursifs suivants. Donner uniquement le contenu de la pile à chaque récursion ainsi que les messages affichés à l'écran.

Hypothèses :

- l'appel est fait avec le tableau ['bonjour','je','suis', 'récursif']
- la procédure $AfficheEcran(str)$ existe et permet d'afficher la chaîne de caractéres str à l'écran

Procedure AffAp(i, tabmots)

Entrée: un entier i initialisé à 0, un tableau de chaînes de caractéres $tabmots$

```
1: Si i < longueur[ tabmots ] Alors
2:     espace ← ' '
3:     AfficheEcran( tabmots[i] )
4:     AfficheEcran( espace )
5:     AffAp( i+1, tabmots )
6: Fin Si
```

Procedure AffAv(i, tabmots)

Entrée: un entier i initialisé à 0, un tableau de chaînes de caractéres $tabmots$

```
1: Si i < longueur[ tabmots ] Alors
2:     espace ← ' '
3:     AffAv( i+1, tabmots )
4:     AfficheEcran( tabmots[i] )
5:     AfficheEcran( espace )
6: Fin Si
```

Principe de la récursivité multiple : couper un gâteau

Manger un gâteau

- gâteau trop gros pour être mangé directement
- on n'arrive à manger qu'une petite part (une bouchée) à la fois
- on sait comment couper le gâteau

⇨ on coupe le gâteau jusqu'à ce que les parts soient suffisamment petites pour que l'on puisse les manger

Résoudre un problème

- problème trop difficile à traiter par une approche classique
- on ne sait résoudre simplement qu'une étape
- on sait comment décomposer le problème

⇨ on décompose le problème jusqu'à ce qu'il soit suffisamment simple pour que l'on puisse le résoudre

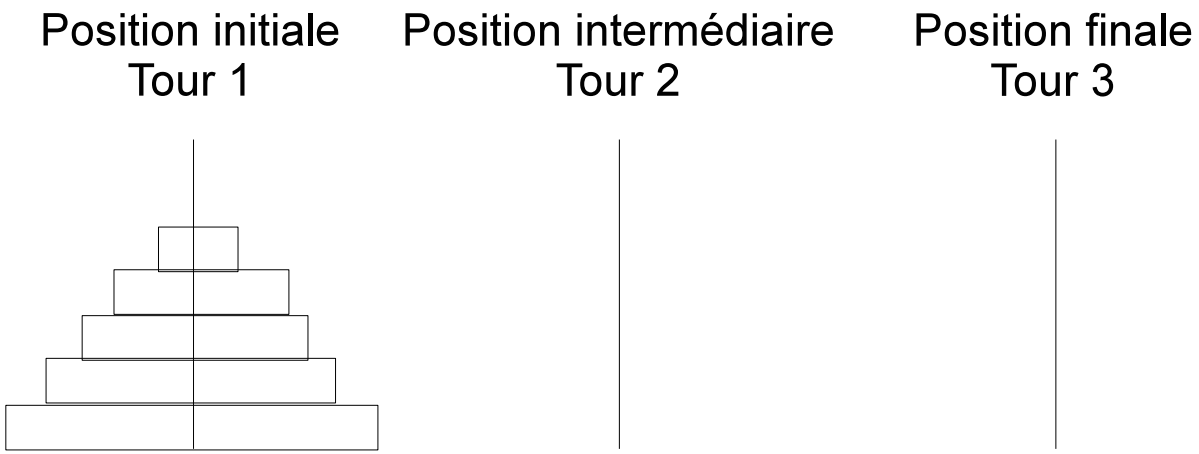
⇨ Approche "diviser pour régner"

Principe de la récursivité multiple : couper un gâteau

Exemple du problème des tours de Hanoï

- Transférer n disques (les un après les autres) de l'axe 1 à l'axe 3, en utilisant B, de sorte que jamais un disque ne repose sur un disque de plus petit diamètre.

$n = 5$



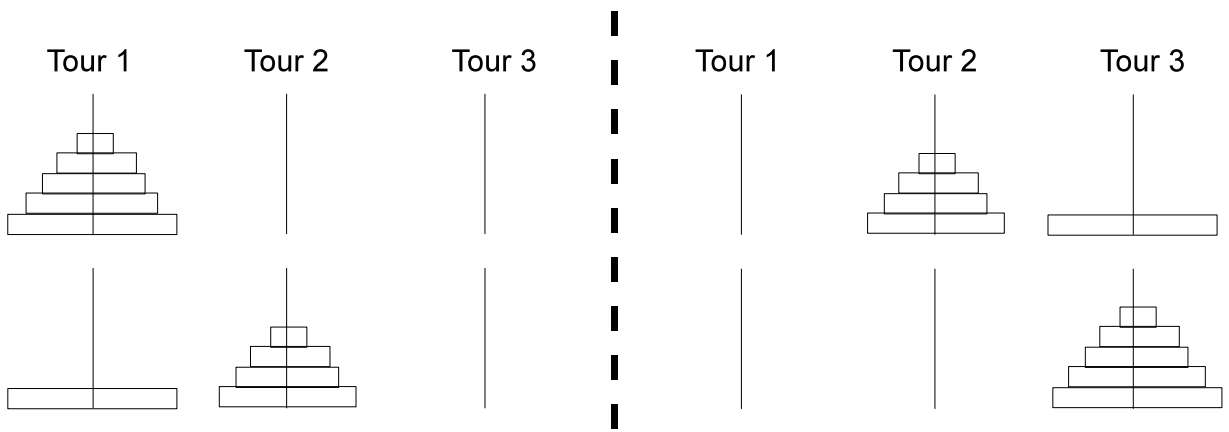
Principe de la récursivité multiple : couper un gâteau

Problème des tours de Hanoï difficile a priori, mais on sait :

- résoudre ce problème si $n-1$ disques sont déjà sur la tour intermediaire
- déplacer un disque

Solution :

- déplacer d'abord $n-1$ disques de la tour initiale vers la tour intermédiaire (en respectant la contrainte sur la taille des disques)
- déplacer le disque n de la tour initiale vers la tour finale
- déplacer $n-1$ disques de la tour intermédiaire vers la tour finale



Principe de la récursivité multiple : couper un gâteau

Algorithme résolvant le problème des tours de Hanoï

Procedure Hanoi(n, départ, final, intermédiaire)

Entrée: $n > 1$ un entier

```
1: Si n = 1 Alors
2:     déplacer( départ , final )
3: Sinon
4:     Hanoi( n-1, départ, intermédiaire, final)
5:     déplacer( départ , final )
6:     Hanoi( n-1, intermédiaire, final, départ )
7: Fin Si
```

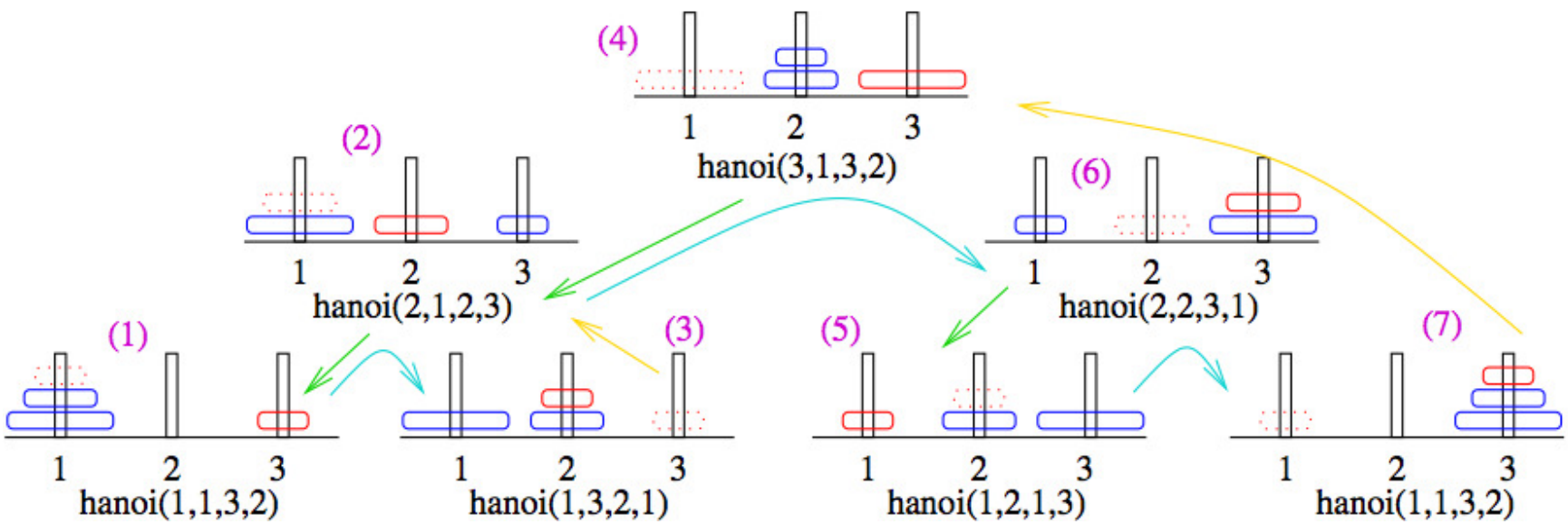
- étape 1 : déplacer les $n-1$ disques de la tour de départ vers la tour intermédiaire ⇨ faire `Hanoi( n-1, départ, intermédiaire, final )`
- étape 2 : déplacer le dernier disque vers la destination finale ⇨ faire `déplacer( départ , final )`
- étape 3 : déplacer les $n-1$ disques de la tour intermédiaire vers la tour finale ⇨ faire `Hanoi( n-1, intermédiaire, final, départ )`

Fonctionnement de la récursivité multiple

Même mécanisme que la récursivité simple : utilisation par l'ordinateur de la pile système

➪ Mais un arbre d'appels récursifs à la place d'une simple chaîne

Exemple : execution de Hanoi(3,1,3,2)



Exercice

La suite de Fibonacci

- un petit peu d'histoire
  - découverte vers 1202 par Léonard de Pise
  - problème de la prolifération des lapins : possédant au départ un couple de lapins, combien de couples de lapins obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?
- la formule : $u_1 = u_2 = 1$, et pour tout n entier : $u_n = u_{n-1} + u_{n-2}$

Ecrire l'algorithme permettant de calculer la suite de Fibonacci pour un entier n en entrée.

Donner la trace de l'exécution de cet algorithme (sous forme d'arbre des appels récursifs) pour $n = 4$. Pour chaque appel, vous donnerez également le contenu de la pile systéme.

Précautions à prendre

Attention aux appels récursifs infinis ....

Fonction F(n)

Entrée: un entier $n > -2$
1: **Retourner** F(n-1)

Fonction P(n)

Entrée: un entier $n > 0$
1: **Si** n = 0 **Alors**
2: **Retourner** 1
3: **Fin Si**
4: **Retourner** P(n+1)

- Définir une condition terminale
  - une condition terminale = un cas particulier pour lequel il n'y a pas d'autre appel récursif
  - p.ex. ajouter au début de F : **Si** n= -1 **Alors Retourner** 0 **Fin Si**
- Vérifier la terminaison de l'algorithme
  - aucune solution automatique pour vérifier la terminaison d'un algorithme
  - regarder au cas par cas (l'utilisation de la récurrence peut aider)

Ecrire un algorithme récursif

Un algorithme récursif doit comporter :

- un cas d'arrêt dans lequel aucun autre appel n'est effectué
  - conseil : la mettre au début de l'algorithme récursif
- un cas général dans lequel un ou plusieurs autres appels sont effectués
  - l'enchaînement des appels doit conduire au critére d'arrêt

---

Fonction Fact(n)

Entrée: un entier $n \geq 0$
Sortie: l'entier $n!$

```
1: Si n = 0 Alors
2:     Retourner 1
3: Fin Si
4: res ← Fact( n-1) . n
5: Retourner res
```

Récursif ou itératif ?

Les algorithmes itératifs peuvent toujours s'écrire sous forme récursive et inversement (mais dans ce dernier cas il faut certaines fois recoder l'utilisation d'une pile).

Avantage de la récursivité :

- simplicité d'expression des algorithmes
- parfois pas d'autres solutions

Inconvénient de la récursivité :

- des précautions à prendre
- peut être plus coûteux
  - taille mémoire de la pile
  - opérations sur la pile

Conseils :

- à utiliser pour des problèmes typiquement récursifs, ne pouvant être résolus de façon itérative
- éviter d'utiliser la récursivité lorsqu'on peut la remplacer par une définition itérative, à moins de bénéficier d'un gain considérable en simplicité.

Application aux algorithmes de tri

Problématique :

- étant donné une structure linéaire (tableau, liste,...) contenant des valeurs d'un type ordonné (entiers, réels, chaîne de caractères, ...), il faut trier les éléments en ordre croissant (ou décroissant)

Des dizaines d'algorithmes répondant à ce problème, mais des approches très différentes en fonction des algorithmes

- itératif, récursifs, ...

Tri sélection

Principe :

- parcourir le tableau pour trouver le plus grand élément, une fois arrivée au bout du tableau placer cet élément par permutation, et recommencer sur le reste du tableau

---

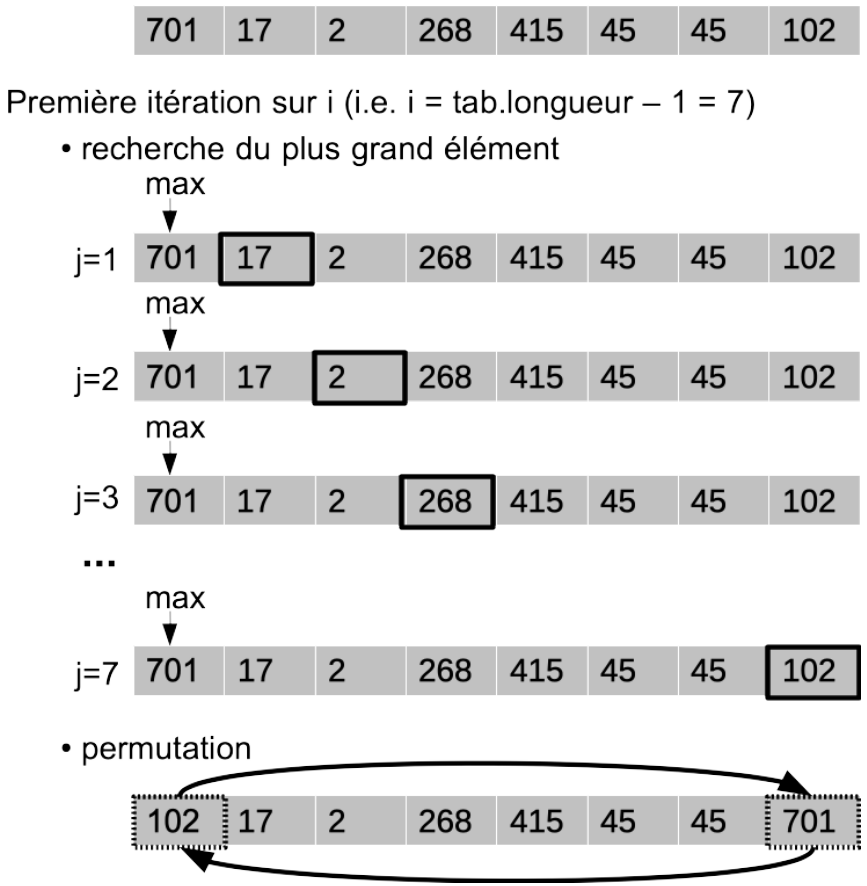
Procedure TriSelection(element[] tab)
Entrée: un tableau d'éléments à trier
Sortie: le tableau trié

```
1:  i ← tab.longueur − 1
2:  Tant que i > 0 faire
3:      max ← 0
4:      Pour j de 1 à i faire
5:          Si tab[j] > tab[max] Alors
6:              max ← j
7:          Fin Si
8:      Fin Pour
9:      temp ← tab[max]
10:     tab[max] ← tab[i]
11:     tab[i] ← temp
12:     i ← i − 1
13: Fin Tant que
```

Remarque : i représente la dernière case à traiter

Tri sélection

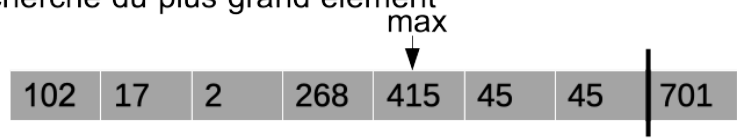
Exemple : trier en ordre croissant le tableau



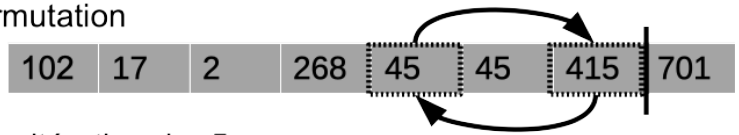
Tri sélection

Deuxième itération i = 6

- recherche du plus grand élément

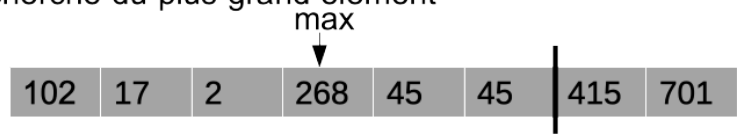


- permutation

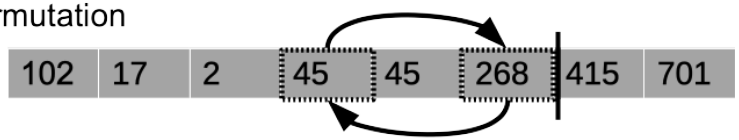


Troisième itération i = 5

- recherche du plus grand élément



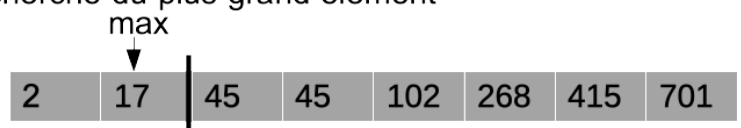
- permutation



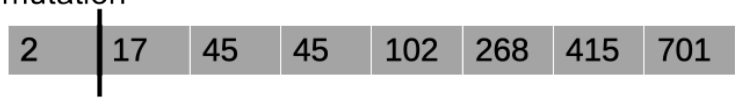
...

Septième itération i = 1

- recherche du plus grand élément



- permutation



Tri fusion

Principe :

- découper le tableau en deux, trier chacune des parties (appels récursifs), puis fusionner
- algorithme récursif avec pour cas d'arrêt "si le tableau a un élément, il est trié"

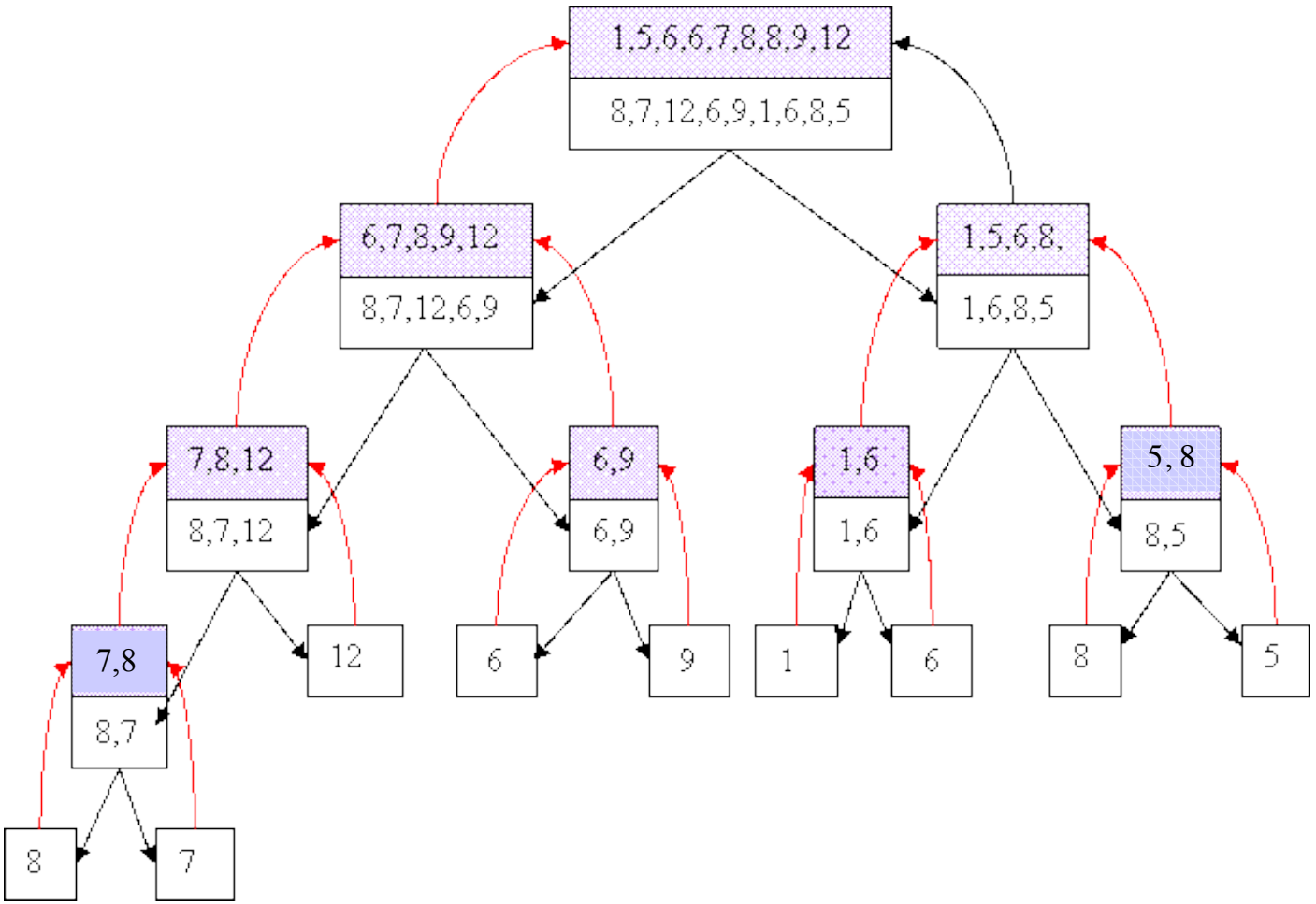
Méthode principale

Procedure triFusion(element[] tab)
Entrée: un tableau d'éléments à trier
Sortie: le tableau trié
1: mergeSort(tab, 0, longueur(tab)-1)

Méthode récursive

Procedure mergeSort(element[] tab, entier i, entier j)
Entrée: une partition du tableau tab située entre les indices i et j
Sortie: la partition triée
1: **Si** $i < j$ **Alors**
2: mergeSort(tab, i, (i+j)/2)
3: mergeSort(tab, (i+j)/2+1, j)
4: fusion(tab, i, (i+j)/2, j)
5: **Fin Si**

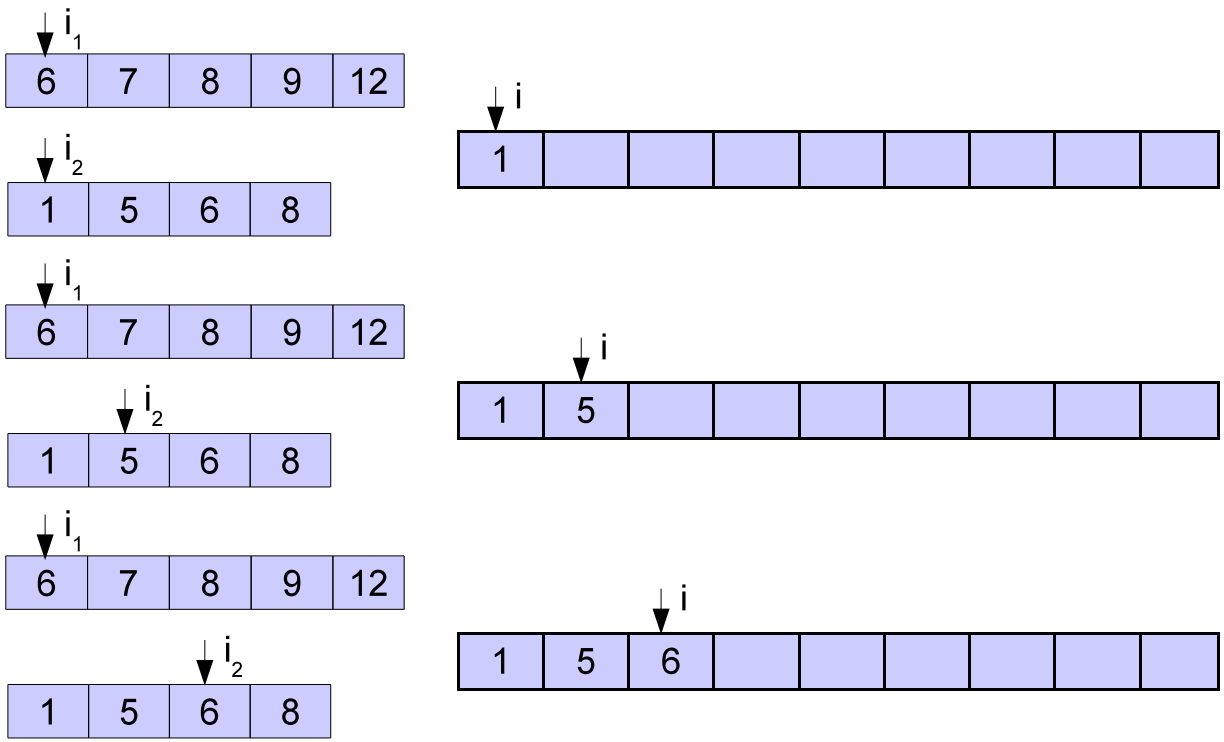
Tri fusion


1,5,6,6,7,8,8,9,12
8,7,12,6,9,1,6,8,5
6,7,8,9,12
8,7,12,6,9
1,5,6,8,
1,6,8,5
7,8,12
8,7,12
6,9
6,9
1,6
1,6
5, 8
8,5
7,8
8,7
12
6
9
1
6
8
5
8
7

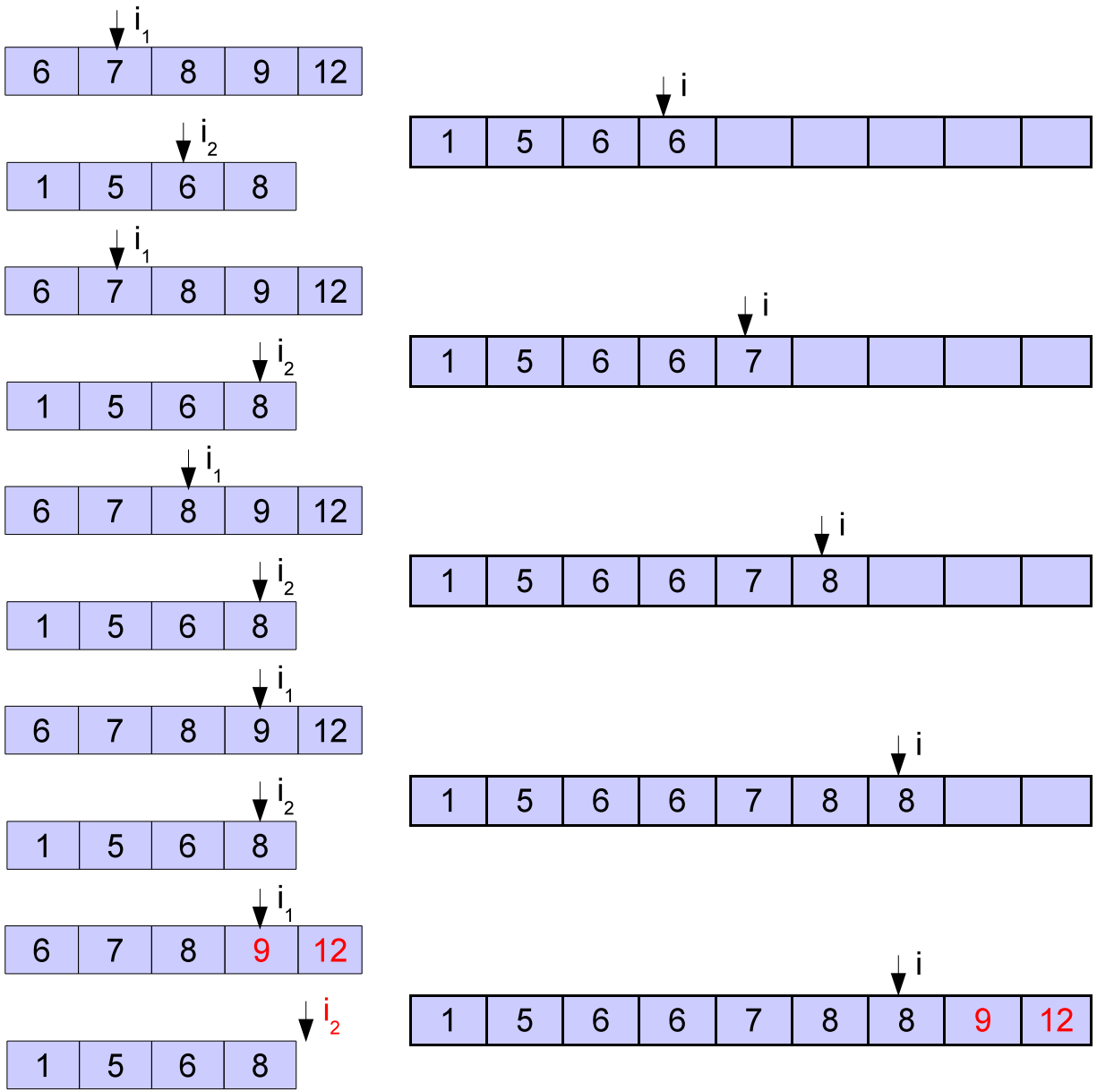
Tri fusion

Principe de la fusion :

- prend en entrée deux tableaux triés et réunis les éléments dans un tableau résultat de telle sorte que celui-ci soit trié
- parcourt les deux tableaux en parallèle, et insère au fur et à mesure les éléments dans le tableau résultat



Tri fusion


i1
6 7 8 9 12
i2
1 5 6 8
i1
6 7 8 9 12
i2
1 5 6 8
i1
6 7 8 9 12
i2
1 5 6 8
i1
6 7 8 9 12
i2
1 5 6 8
i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6
i
1 5 6 6 7
i
1 5 6 6 7 8
i
1 5 6 6 7 8 8
i
1 5 6 6 7 8 8 9 12

Tri fusion

Procedure fusion(element[] T, element[] T1, element[] T2, entier n1, entier n2)

Entrée: **T le tableau résultat, T1 et T2 deux tableaux, de taille n1 et n2 , triés à fusionner**
Sortie: **le tableau T fusionné et trié**

1: $i \leftarrow 0$; $i_1 \leftarrow 0$; $i_2 \leftarrow 0$
2: **Tant que** $i_1 < n1$ **et** $i_2 < n2$ **faire**
3: **Si** $T1[i_1] < T2[i_2]$ **Alors**
4: $T[i] \leftarrow T1[i_1]$
5: $i_1 \leftarrow i_1 + 1$
6: **Sinon**
7: $T[i] \leftarrow T2[i_2]$
8: $i_2 \leftarrow i_2 + 1$
9: **Fin Si**
10: $i \leftarrow i + 1$
11: **Fin Tant que**
12: **Si** $i_1 < n1$ **Alors**
13: **Pour** j **de** i_1 **à** $n1 - 1$ **faire**
14: $T[i] \leftarrow T1[j]$
15: $i \leftarrow i + 1$
16: **Fin Pour**
17: **Sinon Si** $i_2 < n2$ **Alors**
18: **Pour** j **de** i_2 **à** $n2 - 1$ **faire**
19: $T[i] \leftarrow T2[j]$
20: $i \leftarrow i + 1$
21: **Fin Pour**
22: **Fin Si**

Tri fusion

Amélioration de la fusion : plutôt que de passer trois tableaux en paramètres, travailler sur le tableau initial

Procedure fusion(element[] T, entier deb, entier mid, entier fin)

Entrée: **T le tableau initial triés entre deb et mid, et entre mid+1 et fin**
Sortie: **le tableau T trié entre deb et fin**

1: $i \leftarrow 0$; $i_1 \leftarrow deb$; $i_2 \leftarrow mid + 1$
2: **Tant que** $i_1 \leq mid$ **et** $i_2 \leq fin$ **faire**
3: **Si** $T[i_1] < T[i_2]$ **Alors**
4: $temp[i] \leftarrow T[i_1]$
5: $i_1 \leftarrow i_1 + 1$
6: **Sinon**
7: $temp[i] \leftarrow T[i_2]$
8: $i_2 \leftarrow i_2 + 1$
9: **Fin Si**
10: $i \leftarrow i + 1$
11: **Fin Tant que**
12: **Si** $i_1 < mid + 1$ **Alors**
13: **Pour** j **de** i_1 **à** mid **faire**
14: $temp[i] \leftarrow T[j]$
15: $i \leftarrow i + 1$
16: **Fin Pour**
17: **Sinon Si** $i_2 < fin + 1$ **Alors**
18: **Pour** j **de** i_2 **à** fin **faire**
19: $temp[i] \leftarrow T[j]$
20: $i \leftarrow i + 1$
21: **Fin Pour**
22: **Fin Si**
23: $k \leftarrow 0$
24: **Pour** i **de** deb **à** fin **faire**
25: $T[i] \leftarrow temp[k]$; $k \leftarrow k + 1$
26: **Fin Pour**

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Programmation avancée et Complexité

Chapitre 2 – De Python au langage C

Frédéric Flouvat

Basé sur le cours "C for Python Programmers" de C. Burch (Hendrix College) et E. Patitsas (University of Toronto)

Université de la Nouvelle-Calédonie

frederic.flouvat@univ-nc.nc

Plan

- De Python au langage C
  - Les bases
  - Les principales opérations et instructions
  - Les librairies
  - Les pointeurs

Les bases

Un petit peu d'histoire sur le C

- Créé dans les années 70 par Ken Thompson (Bell Lab.) pour aider au développement du système d'exploitation UNIX.
  - Conçu pour pouvoir accéder à toutes les fonctionnalités des systèmes UNIX
- Le premier langage de développement dans les années 80.
- A influencé de nombreux langages de programmation tels que C++, C#, Java, JavaScript, etc.

Et maintenant ?

- Toujours un langage très utilisé
  - Au coeur des principaux systèmes d'exploitation (Windows, Linux, Mac, Android et iOS);
  - Au coeur des principaux Systèmes de Gestion de Bases de Données (p.ex. Oracle, SQL Server et PostgreSQL);
  - Au coeur des systèmes embarqués et objets connectés.

- Pourquoi cette popularité ?
  - Possibilité de manipuler très finement la mémoire (lectures et écritures);
  - Une utilisation contrôlée des ressources (p.ex. libération de la mémoire);
  - La taille du code généré;

Compilateur versus Interpréteur

- Le compilateur en C
  - Traduit le code (fichier texte) en langage machine (fichier exécutable)
  - Les étapes à suivre:
    - écrire un fichier texte (extensions .c ou .h) contenant le programme en langage C
    - compiler le "code source" en "exécutable"
    - lancer le fichier exécutable

- L'interpréteur en Python
  - Lit le code source et l'exécute directement
  - Les étapes à suivre:
    - écrire un fichier texte (extension .py) contenant le programme en langage C
    - passer ce code source en paramètre de l'interpréteur

➢ Un fonctionnement très différent
  - Une étape en moins pour le Python
  - Un exécutable en général beaucoup plus performant en C

Exemple : mon premier programme en C

Toute instruction en C doit être définie à l'intérieur d'une **fonction/procédure**

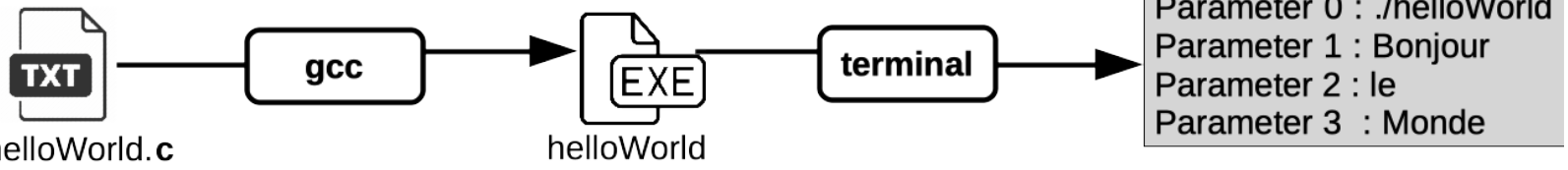
helloWorld.c

```
#include <stdio.h>

int main( int argc, char * argv[] ) {

      int i = 0 ;

      while ( i < argc ) {
            printf("Parameter %d : %s \n", i, argv[i] );
            i++;
      }
      return 0 ;
}
```

Le point d'entrée à l'exécution est la fonction **main**( )

Compilation :
gcc -o helloWorld helloWorld.c

Exécution :
./helloWorld Bonjour le Monde



helloWorld.**c** helloWorld

La déclaration des variables

- Obligation d'indiquer au compilateur les variables avant qu'elles soient utilisées

- En C, déclaration d'une variable = définition du type de la variable
  - Syntaxe : *typeName variableName* ;
  - p.ex.

```
int i ;
double j = 3.2 ;
```

- Impossible de changer le type de la variable après la déclaration

```
int i ;
char i ;
```

```
double j = 3 ;
```

➢ `j` à pour valeur le double 3 .0

- Variable non déclarée ➔ Erreur "symbol undeclared" à la compilation

➢ Avantage : erreur d'utilisation des variables toutes connues à la compilation
(et non au fur et à mesure de l'exécution comme en Python)

Les espaces, tabulations et retours à la ligne

- En **Python**, importants car
  - Retour à la ligne = une nouvelle instruction
  - Espace et tabulation = un bloc (i.e. regroupement) d'instructions

- En **C**, utilisés pour faire de l'indentation (mise en page du code)
  - Fortement conseillé mais non obligatoire
  - Une nouvelle instruction = point virgule **;**
  - Un bloc d'instructions = accolades ouvrante et fermante **{ … }**

Attention : code illisible

C
```
disc=b*b-4*a*c;if(disc<0){
num_sol=0;}else{t0=-b/a;if(
disc==0){num_sol=1;sol0=t0/2
;}else{num_sol=2;t1=sqrt(disc/a;
sol0=(t0+t1)/2;sol1=(t0-t1)/2;}}
```

Python
```
disc = b * b - 4 * a * c
if disc < 0:
    num_sol = 0
else:
    t0 = -b / a
    if disc == 0:
        num_sol = 1
        sol0 = t0 / 2
    else:
        num_sol = 2
        t1 = disc ** 0.5 / a
        sol0 = (t0 + t1) / 2
        sol1 = (t0 - t1) / 2
```

C
```
disc = b * b - 4 * a * c;
if (disc < 0)
{
    num_sol = 0;
}
else
{
    t0 = -b / a;
    if (disc == 0)
    {
        num_sol = 1;
        sol0 = t0 / 2;
    }
    else
    {
        num_sol = 2;
        t1 = sqrt(disc) / a;
        sol0 = (t0 + t1) / 2;
        sol1 = (t0 - t1) / 2;
    }
}
```

La fonction `printf()`

- Affichage de texte à l'écran (la sortie standard)
  - Important notamment pour débugger
  - Un fonctionnement qui peut sembler complexe

- Appartient à la librairie `stdio.h`

- Signature: `int printf ( const char * format, ... );`
  - *`format`* : chaîne de caractères indiquant le format de ce qui va être affiché
    - %d : un entier; %c : un caractère; %s : une chaîne de caractère;
  - *autres paramètres* : les valeurs à afficher
  - cf. http://www.cplusplus.com/reference/cstdio/printf/ pour plus d'informations

```
printf("Parameter");
printf("Parameter %d", 10 );
printf("Parameter %d", i );
printf("Parameter %d \n", i );
printf("Parameter %d : %s \n", i, argv[i] );
```

Les fonctions en C

- Tout code doit être dans une fonction
- Les fonctions ne peuvent être imbriquées
  - Une liste de fonctions pouvant s'invoquer entre elles

C
```c
double expon(double b, int e)
{
    if (e == 0)
    {
        return 1.0;
    }
    else
    {
        return b * expon(b, e - 1);
    }
}
```

- Syntaxe: *typeReturn functionName(type1 paremeter1, type2 parameter2, ⋯){ ⋯ }*
  - Si la fonction ne retourne rien , *typeReturn* = *void*
- Fonction utilisée si invoquée (directement ou indirectement) dans la fonction "main" (le programme principal)

Python
```python
def gcd(a, b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)

print("GCD: " + str(gcd(24, 40)))
```

équivalents ⟺

C
```c
int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

int main()
{
  printf("GCD: %d\n",
    gcd(24, 40));
  return 0;
}
```

Les principales opérations et instructions

Les opérateurs

- Un "mot-clé" du langage utilisé pour faire une opération arithmétique
  - Les principaux opérateurs :

C

```
++ -- (postfixe/préfixe)
+ - ! (unaire)
* / %
+ - (binaire)
< > <= >=
== !=
&&
||
= += -= *= /= %=
```

Python

```
**
+ - (unaire)
* / % //
+ - (binaire)
< > <= >= == !=
not
and
or
```

  - Différences :
    - pas d'opérateur puissance en C (fonction `pow()`)
    - syntaxe différente des opérateurs logiques
    - affectation : opérateur en C vs instruction en Python
      - possibilité de faire une affectation dans une instruction

```
while( (a = getchar()) != EOF )…
```

    - opérateurs ++, --, +=, -=, *=, …
    - division en C : division entière si les deux valeurs en entrée sont entières
      - 13/5 = 2

Les principaux types (1/2)

Les types entiers

Type	Taille	Valeurs
`char`	1 octet	-128 à 127 ou 0 à 255
`unsigned char`	1 octet	0 à 255
`signed char`	1 octet	-128 à 127
`int`	2 ou **4 octets***	-32 768 à 32 767 ou **-2 147 483 648 à 2 147 483 647**
`unsigned int`	2 ou **4 octets***	0 à 65 535 ou **0 à 4 294 967 295**
`short`	2 octets	-32 768 à 32 767
`unsigned short`	2 octets	0 à 65 535
`long`	8 octets	-9223372036854775808 à 9223372036854775807
`unsigned long`	8 octets	0 à 18446744073709551615

*dépend du compilateur (fonction sizeof(type) pour connaître la taille)

Les types réels

Type	Taille	Valeurs	Précision
`float`	4 octets	$3.4*10^{-38}$ à $3.4*10^{+38}$	simple précision (7 chiffres significatifs)
`double`	8 octets	$1.7*10^{-308}$ à $1.7*10^{+308}$	double précision (15 chiffres significatifs)
`long double`	10 octets	$3.4*10^{-4932}$ à $3.4*10^{+4932}$	double précision au moins

Les principaux types (2/2)

- Pas de type booléen en C
  - Utilisation d'un entier
    - 0 = *false* et 1 = *true*

```
int main()
{
    int i = 5;
    if (i)
    {
        printf("in if\n");
    }
    else
    {
        printf("in else\n");
    }
    return 0;
}
```

valide mais code difficile à lire

```
int main()
{
    int i = 5;
    if (i != 0)
    {
        printf("in if\n");
    }
    else
    {
        printf("in else\n");
    }
    return 0;
}
```

- Traités comme des entiers par les opérateurs
  - p.ex. nombre de variables positives

```
pos = (a > 0) + (b > 0) + (c > 0);
```

Remarque sur les accolades { et }

- Indique un bloque (i.e. groupe) d'instructions ➔ pas obligatoire si une seule instruction

```
if (first > second)
    max = first;
else
    max = second;
```

```
if (first > second)
{
    max = first;
}
else
{
    max = second;
}
```

```
if (disc < 0)
{
    num_sol = 0;
}
else
{
    if (disc == 0)
    {
        num_sol = 1;
    }
    else
    {
        num_sol = 2;
    }
}
```

```
if (disc < 0)
{
    num_sol = 0;
}
else
    if (disc == 0)
    {
        num_sol = 1;
    }
    else
    {
        num_sol = 2;
    }
```

```
if (disc < 0)
{
    num_sol = 0;
}
else if (disc == 0)
{
    num_sol = 1;
}
else
{
    num_sol = 2;
}
```

```
if (disc < 0)
    num_sol = 0;
else if (disc == 0)
    num_sol = 1;
else
    num_sol = 2;
```

- Conseil: laisser tout de même les accolades
  - Facilite l'ajout d'instructions
  - Facilite la lecture du code

Synthèse des principales instructions (1/2)

- Les déclarations de variable `int x;`
- Les expressions `x = y + z ;` `printf("%d", x) ;`
- Les conditions `if ( x < 0 ) { printf(" negative"); }`
- Le `return` `return 0 ;`
  - `void` si pas de valeur retournée
- Les boucles `while(`*`condition`*`) { … }`
- Les boucles `for(`*`init; test; update`*`)`

```
for (p = 1; p <= 512; p *= 2) {
    printf("%d\n", p);
}
```

```
while (i >= 0)
{
    printf("%d\n", i);
    i--;
}
```

```
for (i = 0; i < n; i++)
{
    …
}
```

Synthèse des principales instructions (2/2)

- Autre formes de conditions : `switch`
  - Remplace `if … else if … else if … else`
  - `case` : un caractère ou un entier
  - Attention : ne pas oublier les `break`
    - sinon exécution de tous les blocs

```
switch (letter_grade) {
case 'A':
    gpa += 4;
    credits += 1;
    break;
case 'B':
    gpa += 3;
    credits += 1;
    break;
case 'C':
    gpa += 2;
    credits += 1;
    break;
case 'D':
    gpa += 1;
    credits += 1;
    break;
case 'W':
    break;
default:
    credits += 1;
}
```

Les structures de données (1/2)

- Peu de structure de données directement disponibles en C (contrairement à Python)
- Les structures composées
  - Regroupement de variables

```
struct Coordonnees
{
    int x; // Abscisses
    int y; // Ordonnées
};

int main( int argc, char * argv[] )
{
    struct Coordonnees point ;
    point.x = 0 ;
    point.y = 1 ;
};
```

  - Possibilité de définir un type en combinant avec `typedef`

```
struct Coord
{
    int x; // Abscisses
    int y; // Ordonnées
};

typedef struct Coord Coordonnees ;
```

OU

```
typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;
```

```
int main( int argc, char * argv[] )
{
    Coordonnees point ;
    point.x = 0 ;
    point.y = 1 ;
};
```

Les structures de données (2/2)

- Les tableaux
  - Similaires aux listes Python mais avec une taille fixe défini à la création de la variable

```c
char vowels[6] = {'a','e','i','o','u','y'};
```

```c
double pops[50];
pops[0] = 897934;
pops[1] = pops[0] + 11804445;
```

  - Pas d'accès direct à la taille du tableau après sa création
    - pas de `len(vowels)`
    - `size_t n = sizeof(vowels) / sizeof(char);`

- Attention si utilisation d'un indice hors des limites du tableau
  - Python : arrête le programme "proprement" en indiquant l'erreur
  - C : accède à la case mémoire en question
    - pas d'erreur : le programme fonctionne avec des mauvaises valeurs
    - erreur avec arrêt du programme (*segmentation fault* ou *bus error*) mais pas forcément à l'endroit du code où l'accès a été effectué
    - très difficile à débugger

Les librairies

Les librairies et prototypes de fonction

- Organiser son code en le répartissant dans plusieurs fichiers
  - Améliore la lisibilité, la maintenance, l'évolutivité du code

- Problème : fonctions déclarées obligatoirement avant leur utilisation en C
- Comment faire quand les fonctions sont déclarées dans des fichiers différents ?
  - Un vrai problème pour le compilateur

- Les prototypes de fonction
  - Décrire uniquement l'entête de la fonction (sa signature), sans son corps

main.c

```
int gcd(int a, int b);

int main()
{
    printf("GCD: %d\n", gcd(24, 40));
    return 0;
}
```

indique au compilateur que la fonction n'a pas encore été définie

```
int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}
```

math.c

Les fichiers entêtes ou *header*

- Un fichier avec l'extension *.h* regroupant des prototypes de fonction (et la définition des structures de données)
  - Importé par les fichiers sources utilisant la fonction (`#include "…"`)
  - ➢ Evite de répéter les prototypes de fonctions dans chaque fichier source l'utilisant

main.c

```c
#include <stdio.h>
#include "mathfun.h"

int main()
{
    printf("GCD: %d\n", gcd(24, 40));
    return 0;
}
```

mathfun.h

```c
int gcd(int a, int b);

double expon( double  b, int e );

…
```

mathfun.c

```c
#include "mathfun.h"

int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

double expon(double b, int e)
{
    if (e == 0)
        return 1.0;
    else
        return b * expon(b, e - 1);
}

…
```

Le préprocesseur

- Programme prétraitant le code source avant de l'envoyer au compilateur
  - Langage basé sur des commandes (*directives*)

➢ Inclusion de fichiers : `#include "filename"` ou `#include <filename>`
- remplace le "*include*" par le contenu du fichier associé (i.e. les prototypes de fonctions)

➢ Définition de macros : `#define VARNAME value`
- des constantes (texte remplacé avant la compilation)

```
#define PI 3.14159
```

```
printf("area: %f\n", PI * r * r);
```

➢ Compilation conditionnelle : `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` et `#endif`
- p.ex. inclure des librairies différentes en fonction de l'OS ciblé (paramètre du compilateur)

```
#ifdef __unix__
#include <unistd.h>
#elif defined _WIN32
#include <windows.h>
#endif
```

Compiler un code réparti entre plusieurs fichiers

1. Insérer des directives de compilation conditionnelle pour éviter au compilateur d'essayer de compiler plusieurs fois le même code (*error: redefinition of function …*)

mathfun.h

```
#ifndef MATHFUN_H
#define MATHFUN_H

int gcd(int a, int b);

double expon( double  b, int e );

…
#endif
```

mathfun.c

```
#include "mathfun.h"

int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}

…
```

2. Compiler chaque fichier source pour créer un fichier objet
   - gcc –c –Wall main.c
   - gcc –c –Wall mathfun.c

3. Faire l'édition des liens et construire l'exécutable final (lier les fichiers compiler pour créer l'exécutable)
   - gcc –o myprog main.o mathfun.o

Compiler un code réparti entre plusieurs fichiers

- Possibilité d'utiliser un fichier **Makefile** pour éviter de tout devoir retaper à chaque fois

makefile

```
myprog: main.o structure.o operation.o
    gcc -o prog main.o structure.o operation.o

main.o : main.c
    gcc -c -Wall main.c

structure.o : structure.c
    gcc -c -Wall structure.c

operation.o : operation.c
    gcc -c -Wall operation.c
```

- Ecrire simplement *make* dans le terminal pour tout recompiler

➢ Cf. http://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/Makefile.html pour plus de détails

Les pointeurs

Les bases

- Un pointeur = une variable représentant l'adresse mémoire vers une donnée
  - Un concept caché dans beaucoup d'autres langages mais sous jacent
- Déclaration : *typeName * variableName ;*
- Accès à l'adresse mémoire d'une variable : operateur &
- Accès à l'emplacement mémoire référencé par un pointeur : opérateur *
  - appelé aussi "déréférencement"
- Pointeur null : NULL

```
int i;
int *p;

i = 4;
p = &i;
*p = 5;
printf("%d\n", i);
```

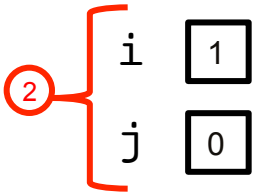
Passage par valeur, par référence et pointeurs (1/3)

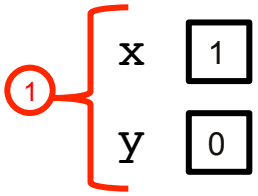
- Passage par valeur d'un paramètre d'une fonction
  - La valeur est recopiée de manière temporaire dans la fonction

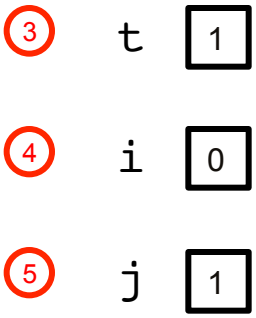
```
void swap(int i, int j) {
    int t;

3   t = i;
4   i = j;
5   j = t;
}
```

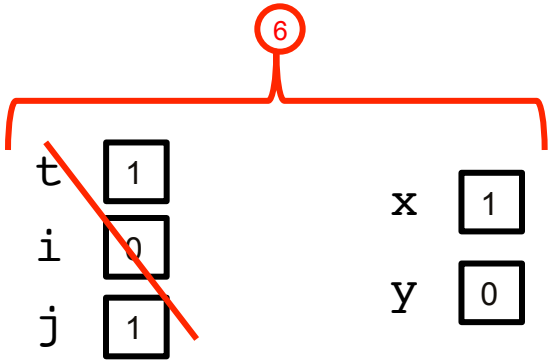
```
1   int x = 1 ;
    int y = 0;

2   swap(x, y);   6

    printf("x=%d, y=%d", x, y);
```

compile mais ne fait rien !









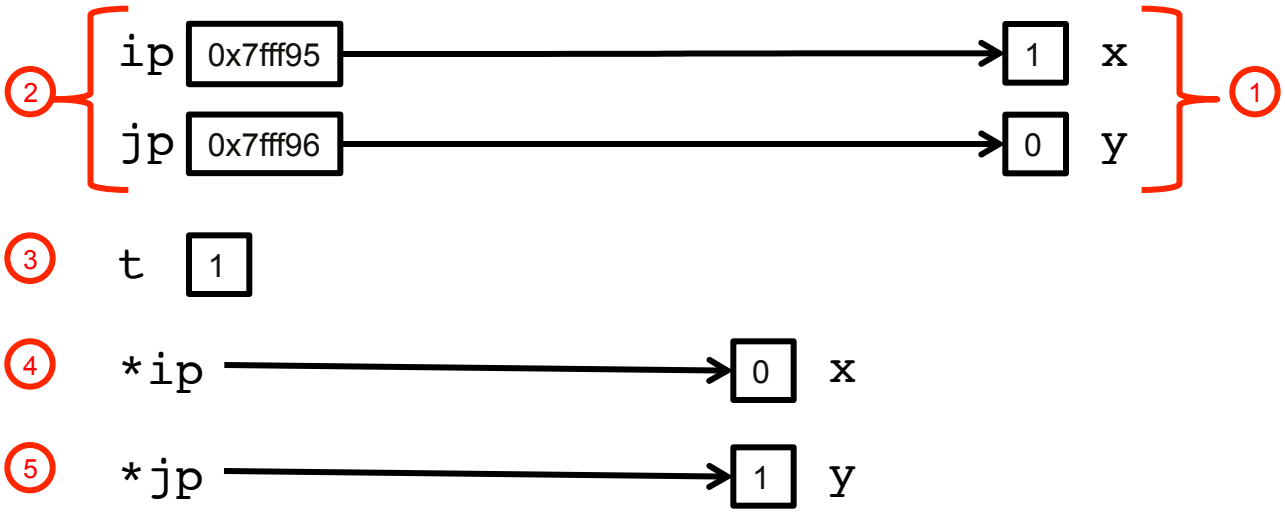
Passage par valeur, par référence et pointeurs (2/3)

- Passage par référence (pointeur) d'un paramètre d'une fonction
  - L'adresse mémoire est recopiée de manière temporaire, mais il est possible d'y accéder est de modifier durablement son contenu

```
void swap(int * ip, int * jp) {
    int t;

3   t = *ip;
4   *ip = *jp;
5   *jp = t;
}
```

```
1   int x = 1 ;
    int y = 0;

2   swap( &x, &y);

    printf("x=%d, y=%d", x, y);
```

échange les valeurs de x et y



Passage par valeur, par référence et pointeurs (3/3)

- **Autre exemple** : la fonction `scanf`
  - Fonction permettant de lire les informations saisies au clavier
  - Prend en deuxième paramètre l'adresse mémoire d'une variable pour pouvoir enregistrer la valeur lue à l'intérieur

```
printf("Type a number. ");
scanf("%d", &i);
printf("The value is %d", i );
```

- **Attention :**

```
void swap(int * ip, int * jp) {
    int * tp;

    tp = ip;
    ip = jp;
    jp = tp;
}
```

```
int x = 1 ;
int y = 0;

swap(&x, &y);

int * xp ;
int * yp ;

xp = &x ;
yp = & y ,

swap(xp, yp);

printf("x=%d, y=%d", x, y);
```

compile mais ne fait rien !

Retour sur les tableaux

- Variable de type tableau en C = un pointeur vers la première case du tableau en mémoire
  - Les autres cases sont à la suite

```
char vowels[6] = {'a','e','i','o','u','y'};
printf("%c", vowels[2] );
```

⟺

```
char vowels[6] = {'a','e','i','o','u','y'};
printf("%c", *(vowels+2) );
```

- Passer en paramètre un tableau à une fonction = passer en paramètre un pointeur

```
void setToZero(int *arr, int n) {
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = 0;
    }
}

int main() {
    int grades[50];

    setToZero(grades, 50);
    return 0;
}
```

Les chaînes de caractères (*string*)

- Pas de type prédéfini ➔un simple tableau de caractères (i.e. un pointeur)
  - Fin de la chaîne de caractères : caractère `\0`

```
char sentence[20] = "the dog is agog.";
```

t	h	e		d	o	g		i	s		a	g	o	g	.	\0			

- Copier une chaîne de caractères = copier un tableau

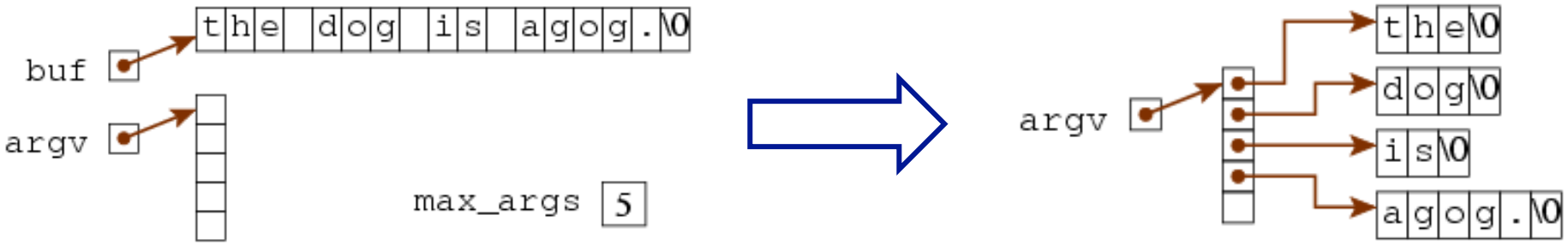
```
for (i = 0; src[i] != '\0'; i++) {
    dst[i] = src[i];
}
dst[i] = '\0';
```

- Beaucoup de fonctions déjà implémentées dans `string.h`
  - **void** strcpy(**char** *dst, **char** *src)
    - copie deux chaînes (dont le \0)
  - **int** strlen(**char** *src)
    - retourne la taille de la chaîne (sans le \0)
  - **int** strcmp(**char** *a, **char** *b)
    - retourne 0 si les deux chaînes sont identiques, <0 si a est plus petit que b et >0 si a est plus grand que b (ordre lexicographique)

Application à l'extraction des mots dans une phrase (*tokenization*)

stringFunc.h

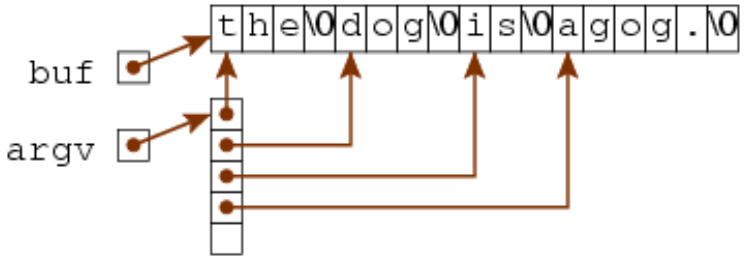
```
#indef STRINGFUNC
#define STRINGFUNC

#include <ctype.h>

/* splitLine
 *  Breaks a string into a sequence of words. The pointer to each successive word
 *  is placed into an array. The function returns the number of words found.
 *
 * Parameters:
 *  buf - string to be broken up
 *  argv - array where pointers to the separate words should go.
 *  max_args - maximum number of pointers the array can hold.
 *
 * Returns:
 *  number of words found in string.
 */
int splitLine(char *buf, char **argv, int max_args) ;

#endif
```



Application à l'extraction des mots dans une phrase (*tokenization*)

stringFunc.c

```
#include "stringFunc.h"

/* splitLine
 *  Breaks a string into a sequence of words. The pointer to each successive word
 *  is placed into an array. The function returns the number of words found.
 *
 * Parameters:
 *  buf - string to be broken up
 *  argv - array where pointers to the separate words should go.
 *  max_args - maximum number of pointers the array can hold.
 *
 * Returns:
 *  number of words found in string.
 */
int splitLine(char *buf, char **argv, int max_args) {
    int arg;

    /* skip over initial spaces */
    while (isspace(*buf)) buf++;

    for (arg = 0; arg < max_args && *buf != '\0'; arg++) {
      argv[arg] = buf;

      /* skip past letters in word */
      while (*buf != '\0' && !isspace(*buf)) {
        buf++;
      }

      /* if not at line's end, mark word's end and continue */
      if (*buf != '\0') {
        *buf = '\0';
        buf++;
      }

      /* skip over extra spaces */
      while (isspace(*buf)) buf++;
    }
    return arg;
}
```



Structures de données et pointeurs

- Souvent combinés
  - Pointeur sur une structure de données
  - Tableau de structure de données

```
typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;
```

```
int main() {
    Coordonnees c;

    c.x = 50;
    c.y = 100;

    double dist = distToOrigin( &c );

    return 0;
}
```

➢ Peut éviter de recopier beaucoup de valeurs

```
#include <math.h>

double distToOrigin(struct Coordonnees *p) {
    return sqrt( (*p).x * (*p).x + (*p).y * (*p).y) );
}
```

- `(*p).x` peut aussi être écrit `p->x`

```
#include <math.h>

double distToOrigin(struct Coordonnees *p) {
    return sqrt( p->x * p->x + p->y * p->y );
}
```

Allocation dynamique de la mémoire (1/2)

- Permet d'allouer/libérer la mémoire associée à une variable au moment de l'exécution en fonction des données ou des actions des utilisateurs

- Allocation : fonction `void* malloc (size_t size)`
  - `size` : entier représentant la taille en octet de la mémoire à allouer
    - utiliser la fonction `size_t` **`sizeof`**`( typeName )`

- Libération : fonction `void free(void *ptr)`
  - `ptr` : adresse mémoire de la variable à libérer

```c
#include "struct.h"

int main() {
    int n;
    Coordonnees *arr;

    printf("How many points ? ");
    scanf("%d", &n);
    arr = (Coordonnees *) malloc( n * sizeof(Coordonnees) );

    …
    free( arr ) ;
    return 0;
}
```

struct.h

```c
#ifndef STRUCT_H
#define STRUCT_H

typedef struct
{
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnees;

#endif
```

Allocation dynamique de la mémoire (2/2)

- **Attention :**
  - Ne jamais accéder à la mémoire après l'avoir libérée
  - Ne jamais libérer plusieurs fois la mémoire

```
int *arr;
…
arr = (int*) malloc( n * sizeof(int) );
…
free( arr );
…
free( arr ) ;
```

```
int *arr;
…
arr = (int*) malloc( n * sizeof(int) );
…
free( arr ) ;
…
int j = arr[0];
```

  - Mais la mémoire doit être libérée
  - ➢ Sinon, fuite mémoire

Application à la gestion d'une liste chaînée (1/3)

```c
#include <stdlib.h>

typedef struct node {
    int data;
    struct node *next;
} Node ;

typedef struct {
    Node *first;
} List;

/* listCreate
 *  Creates an empty linked list.
 *
 * Returns:
 *  allocated list, holding nothing.
 */
List* listCreate() {
  List *ret;

  ret = (List*) malloc( sizeof(List) );
  ret->first = NULL;
  return ret;
}

…
```

Application à la gestion d'une liste chaînée (2/3)

```
/* listRemove
 *  Removes first occurrence of number from a
 *  linked list, returning 1 if successful
 *
 * Parameters:
 *  list - list from which item is to be removed.
 *  to_remove - number to be removed from the list.
 *
 * Returns:
 *  1 if item was found and removed, 0 otherwise.
 */
int listRemove(List *list, int to_remove) {
  Node * prev, cur ;

  prev = NULL;
  cur = list->first
  while (cur != NULL) {
   if (cur->data == to_remove) {
     if (prev == NULL) {
       list->first = cur->next;
     } else {
       prev->next = cur->next;
     }
     free(cur);
     return 1;
   }
   prev = cur;
   cur = cur->next;
 }
 return 0;
}
```

Application à la gestion d'une liste chaînée (3/3)

- Une autre implémentation possible de `listRemove` :

```
int listRemove(List *list, int to_remove) {
  Node **cur_p;
  Node *out;
  int cur_data;

  cur_p = &(list->first);
  while (*cur_p != NULL) {
    cur_data = (*cur_p)->data;
    if (cur_data == to_remove) {
      out = *cur_p;
      *cur_p = out->next;
      free(out);
      return 1;
    }
    cur_p = &((*cur_p)->next);
  }
  return 0;
}
```

- curr_p représente un pointeur vers le pointeur du premier nœud
  - i.e. l'adresse en mémoire où est stocké l'adresse du premier nœud

➢ Permet de changer ces adresses et donc le chaînage de la liste

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Programmation avancée et Complexité

Chapitre 3 - Complexité et Ordre de grandeur

Frédéric Flouvat

Université de la Nouvelle-Calédonie

Plan

Un problème, plusieurs solutions

Pour un problème, plusieurs méthodes de résolutions possibles

Ex. problème du calcule de x^n

- *Méthode 1* : $x^n = x^{n-1}.x$
- *Méthode 2* (méthode des facteurs) :

$$x^n = \begin{cases} x & \text{si } n = 1\,; \\ x^{n-1}.x & \text{si } n \text{ premier}\,; \\ (x^p)^{n'} & \text{si } n = p.n' \text{ avec } p \text{ plus petit diviseur premier de } n. \end{cases}$$

- ...

  (dans certains livres d'algorithmique, 26 pages sont consacrés à ce problème)

Un problème, plusieurs solutions

Pour une méthode de résolution, plusieurs algorithmes possibles

- p.ex. pour la méthode 1

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: res ← 1
2: Pour i de 0 à n − 1 faire
3:     res ← res . x
4: Fin Pour
5: Retourner res
```

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 1 Alors
2:     Retourner x
3: Fin Si
4: res ← Puissance(x, n-1) . x
5: Retourner res
```

⇨ Quelle méthode choisir ? Quel algorithme ? Quel(s) critére(s) ?

⇨ analyser la complexité des algorithmes et choisir la solution la plus intéressante

- p.ex. simplicité, efficacité, quantité de données stockées sur le disque, quantité de trafic généré sur le réseau
- simplicité vs efficacité

Complexité d'un algorithme

Comment comparer des algorithmes ? Si deux algorithmes résolvent le même problème, quel est le meilleur ?

➪ *Intuition* : préférer celui qui utilise le moins de ressources machine

- ressources de calculs
- ressources d'espace de stockage

➪ Méthode 1 : Etudes expérimentales

- implémenter l'algorithme dans un langage de programmation (p.ex. C)
- faire fonctionner le programme avec des entrées de taille et de composition différentes
- mesurer le temps d'exécution et l'espace mémoire occupé

Complexité d'un algorithme

problème de la méthode expérimentale :

- nécessiter d'implémenter
- des résultats ne représentant pas toutes les données en entrée
- nécessités d'utiliser le même environnement (matériels et systéme d'exploitation) pour pouvoir comparer

⇨ Méthode 2 : Analyse théorique

- se fait à partir de l'algorithme, non de l'implémentation
- caractérise les performances comme une fonction de n, la taille de l'entrée
- prend en considération toutes les entrées
- indépendant de l'environnement utilisé

Complexité en temps et en espace

Complexité en temps

- l'algorithme s'exécute le plus rapidement possible

⇨ compter le nombre d'opérations élémentaires (car supposé en temps constant)

- opérations arithmétiques, affectation, instruction de contrôle, entrée-sortie ...
- p.ex. une boucle "Pour i de 1 à n faire"
  ⇨ n itérations $\times$ (1 affectation de i + 1 incrementation de i + 1 test de i) , i.e. $3n$ operations (ou $\sum_{i=1}^{n} 3$)

Complexité en espace

- l'algorithme occupe le moins d'espace mémoire possible

⇨ nombre de cellules mémoire utilisées (sauf les données en entrée)

- taille des entrées-sorties, taille des variables temporaires, ...
- p.ex. une variable entiére = 1 cellule

Complexité d'un algorithme dépend de la taille de son entrée

Exemple de complexité d'un algorithme

Procedure MinListe(L)

Entrée: L une liste d'entier
Sortie: affiche le nombre minimum contenu dans L

```
 1: min ← premier ( L )
 2: i ← 1
 3: Tant que  i < longueur( L ) faire
 4:     Si  iéme( L, i ) < min  Alors
 5:         min ← iéme( L, i )
 6:     Fin Si
 7:     i ← i + 1
 8: Fin Tant que
 9: écrire("le minimum est ")
10: écrire( min )
```

Nombre d'opérations élémentaires

Soit n la longueur de la liste L

1 affectation + 1 appel premier(L)
1 affectation
n comparaisons + n appels à longeur()
(n-1) comparaisons + (n-1) appels à iéme()

m affectations + m appels à iéme()

(n-1) affectations + (n-1) additions

1 écriture

1 écriture

Exemple de complexité d'un algorithme

Complexité en temps de l'algorithme

- en supposant que l'appel à chacune des fonctions *premier()*, *longueur()* et *iéme()* est équivalent à une opération élémentaire
- nombre total d'opérations élémentaires :
  $3 + 2n + 2(n-1) + 2m + 2(n-1) + 2 = 6n + 2m + 1$
- valeur de m ? ⇨ dépend de la liste et de son contenu
  - Meilleur cas : $m = 0$ si le premier de la liste est le minimum
  - Pire cas : $m = n - 1$ si les nombres de la liste sont rangé par ordre décroissant
  - Cas moyen : $m = \frac{n-1}{2}$ si les nombres respectent une distribution parfaitement aléatoire

⇨ Complexité des algorithmes peut aussi dépendre de la structure de données utilisée

Complexité en espace de l'algorithme

- un entier pour stocker le minimum min
- un entier i pour savoir où on est dans la liste

Complexité au pire, au meilleur et en moyenne

Dans certains cas, la complexité (en temps et/ou en espace) dépend d'autres paramètres que la taille des données en entrée de l'algorithme

- p.ex. afficher le minimum d'une liste de nombres

⇨ étude de la complexité au pire cas, au cas moyen et au meilleur cas

Complexité au pire : liée au plus grand nombre d'opérations qu'aura à exécuter l'algorithme pour une entrée de taille n

- avantage : donne une borne maximum, pas de mauvais surprise
- inconvénient : peut ne pas refléter le comportement de l'algorithme en général, peut être très rare

- celle généralement utilisée

Complexité au pire, au meilleur et en moyenne

Complexité en moyenne : moyenne des complexités pour un ensemble de jeux de données de taille n représentatifs (en tenant compte de la probabilité d'apparition de chacun)

- avantage : représente le comportement de l'algorithme en général
- inconvénient : pas une bonne indication pour certains jeux de données
- intéressante pour compléter l'information apportée par la complexité au pire cas

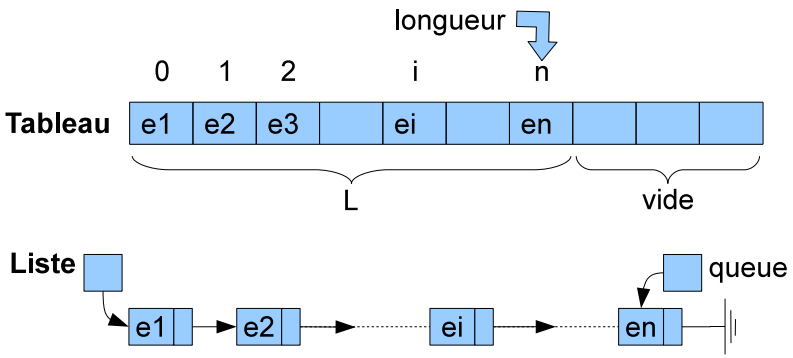
Complexité au meilleur : plus petit nombre d'opérations qu'aura à effectuer l'algorithme pour une entrée de taille n

- peut être un très mauvais indicateur si le meilleur cas est très rare
- peu utilisée pour choisir un algorithme

Attention à l'impact de la structure de données

Un même type abstrait de données peut être implémenté de plusieurs façons

- exemples d'implémentation des listes :
  - par un tableau
    - avantages : simple, économique en espace
    - inconvénients : modifications lentes
  - par une liste chaînée
    - avantages : opérations rapides, gestion souple
    - inconvénients : pas d'accès direct, consomme de l'espace mémoire



⇨ Souvent un fort impact sur l'efficacité des algorithmes

Exercice

Ecrire deux algorithmes permettant de calculer la somme des n premiers entiers et donner leur complexité (temps et espace)

Indication : utiliser pour l'un des algorithmes la propriété mathématique $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

En conclure que suivant l'algorithme, on arrive au même résultat mais pas nécessairement dans le même temps ni en utilisant la même quantité de mémoire.

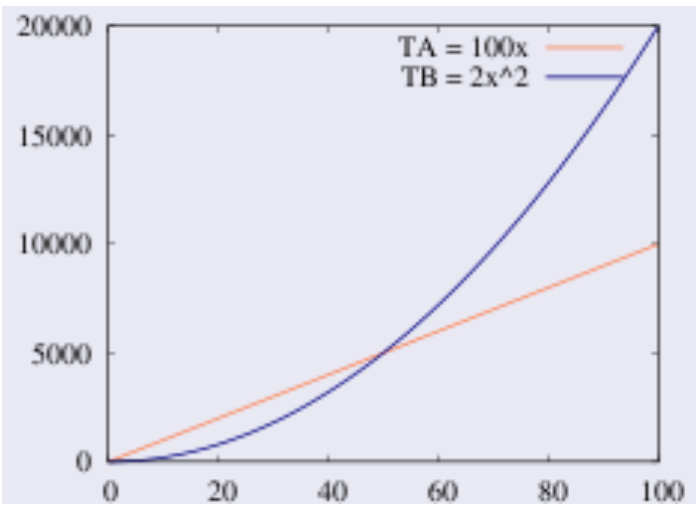
Ordre de grandeur

Jusqu'à présent, étude de la complexité ⇨ compter le nombre d'opérations élémentaires et l'espace occupé

Dans la majorité des cas, connaître le nombre exact d'opérations n'est pas nécessaire, un ordre de grandeur suffit

Illustration de l'inutilité des constantes

- comparaison du temps d'exécution de deux algorithmes A et B
- "temps d'exécution" de A (noté TA) : $T_A(n) = 100n$
- "temps d'exécution" de B (noté TB) : $T_B(n) = 2n^2$



Exemple de la recherche dans un tableau

Recherche dichotomique

- Hypothése : tableau trié
- Principe :
  - comparer la valeur recherchée avec l'élément au milieu du tableau
  - si c'est le même, retourner l'élément du milieu
  - sinon recommencer sur la premiére moitié (ou la deuxiéme) si la valeur recherchée est plus petite (ou plus grande) que l'élément rangé au milieu de la table

Fonction Dicho(chaine[] tab, chaine x)
Entrée: un tableau de chaînes de caractéres
Sortie: une chaîne de caractéres à trouver

```
 1: i ← 0 ; j ← tab.longueur - 1 ;
 2: Tant que i < j faire
 3:     Si tab[(j+i)/2] = x Alors
 4:         Retourner vrai
 5:     Sinon Si tab[(j+i)/2] > x Alors
 6:         j ← (j+i)/2 - 1
 7:     Sinon
 8:         i ← (j+i)/2 +1
 9:     Fin Si
10: Fin Tant que
11: Retourner faux
```

Exemple de la recherche dans un tableau trié

Recherche dichotomique

- Complexité en espace : la place de deux entiers
- Complexité en temps :
  - 2 affectations et 1 retourner + à chaque itération,
    - 1 comparaison d'entiers pour le test de boucle
    - 2 comparaisons de chaînes (1 si l'élément est trouvé)
    - 2 accès à une case d'un tableau
    - 7 opérations arithmétiques + / - (sauf si l'élément est trouvé)
    - 1 affectations $\leftarrow$
  - 1 test de condition d'arrêt du while
  - ⇨ au pire, la longueur du tableau entre i et j est n, puis $n/2$, puis $n/4$, ... , jusqu'à $n/2^t = 1$
  - ⇨ le nombre d'itérations est donc t tel que $n/2^t = 1$, i.e. $2^t = n$ soit $t * log(2) = log(n)$, $t = log_2(n)$
  - ⇨ complexité au pire cas : $13 \times log_2(n) + 4$

Recherche séquentielle

- Principe : examiner successivement tous les éléments de la table jusqu'à ce que l'élément voulu soit trouvé.
- Complexité au pire : $n \times c_s$ avec c_s une constante représentant le coût des opérations à chaque itération

Exemple de la recherche dans un tableau trié

Différence entre recherche dichotomique et séquentielle peut être simplifiée

- dans le pire cas, nombre d'opérations de la recherche dichotomique est proche de $log_2(n)$
- dans le pire cas, nombre d'opérations de la recherche séquentielle est proche de n

⇨ Approximation de la complexité suffisante pour comparer des algorithmes

- on sait qu'il existe une valeur v telle que pour tout $n > v$, $c_d \times log_2(n) < n \times c_s$

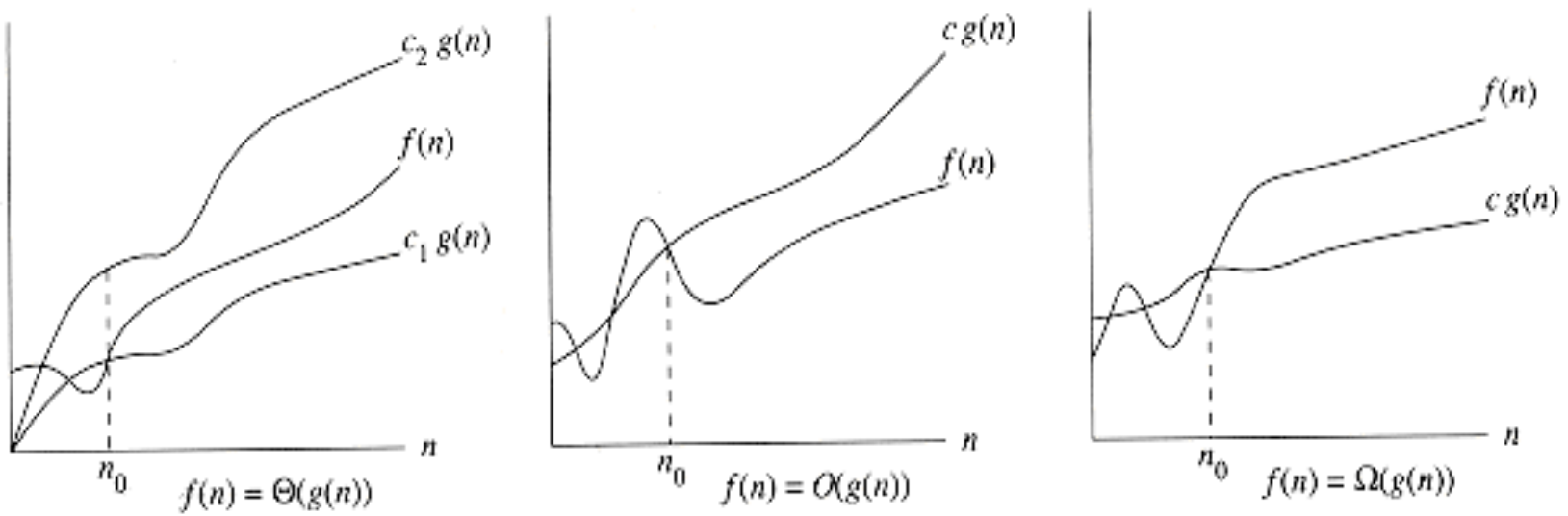
⇨ facilite la comparaison des algorithmes

Notation de Landau

Etude de l'ordre de grandeur

➪ besoin de notations asymptotiques, i.e. de bornes

- O ("grand O") : f en $O(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times g(n)$
  - i.e. majoration
- Ω : f en $\Omega(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, c \times g(n) \leq f(n)$
  - i.e. minoration
- Θ : f en $\Theta(g)$
  $\Leftrightarrow \exists n_0, \exists c_1, c_2 > 0, \forall n \geq n_0, c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$
  - i.e. équivalents à une constante multiplicative près



Notation de Landau : exemples

$2n$ en $O(?)$

- f(n) = 2n et g(n) = ?
- $\exists n_0, \exists c > 0, \forall n \geq n_0, 2n \leq c \times g(n)$
- prenons $g(n) = 3n$
- existe-t- il des valeurs pour n_0 et c telles que $2n \leq c \times 3n$?
- oui, si $n_0 = 1$ et $c = 2/3$

⇨ $2n$ en $O(3n)$

$\frac{1}{2}n^2 - 3n$ en $\Theta(?)$

- f(n) = $\frac{1}{2}n^2 - 3n$ et g(n) = ?
- $\exists n_{0_2}, \exists c_2 > 0, \forall n \geq n_{0_2}, f(n) \leq c_2 \times g(n)$ et $\exists n_{0_1}, \exists c_1 > 0, \forall n \geq n_{0_1}, c_1 \times g(n) \leq f(n)$
- prenons g(n) = n^2
- existe-t- il des valeurs pour n_{0_1}, n_{0_2}, c_1 et c_2 telles que $\frac{1}{2}n^2 - 3n \leq c_2 \times n^2$ et $c_1 \times n^2 \leq \frac{1}{2}n^2 - 3n$?
- oui si $n_{0_2} = 1$, $c_2 = 1/2$, $n_{0_1} = 7$, $c_1 = 1/14$

⇨ $\frac{1}{2}n^2 - 3n$ en $\Theta(n^2)$

Notation de Landau : exercices

n en $O(?)$

- $n_0 = ?$
- $c = ?$

$n^2 - n + 1$ en $\Theta(?)$

- $n_0 = ?$
- $c = ?$

$(n+1)^2$ en $O(?)$

- $n_0 = ?$
- $c = ?$

Comment "deviner" $g(n)$?

Quelques principes généraux en pratique :

- les facteurs constants ne sont pas importants
- les termes d'ordre inférieur sont négligeables
  - $a_k n^k + a_{k-1} n^{k-1} + ... + a_2 n^2 + a_1 n + a_0$ est en $O(n^k)$

Exemples :

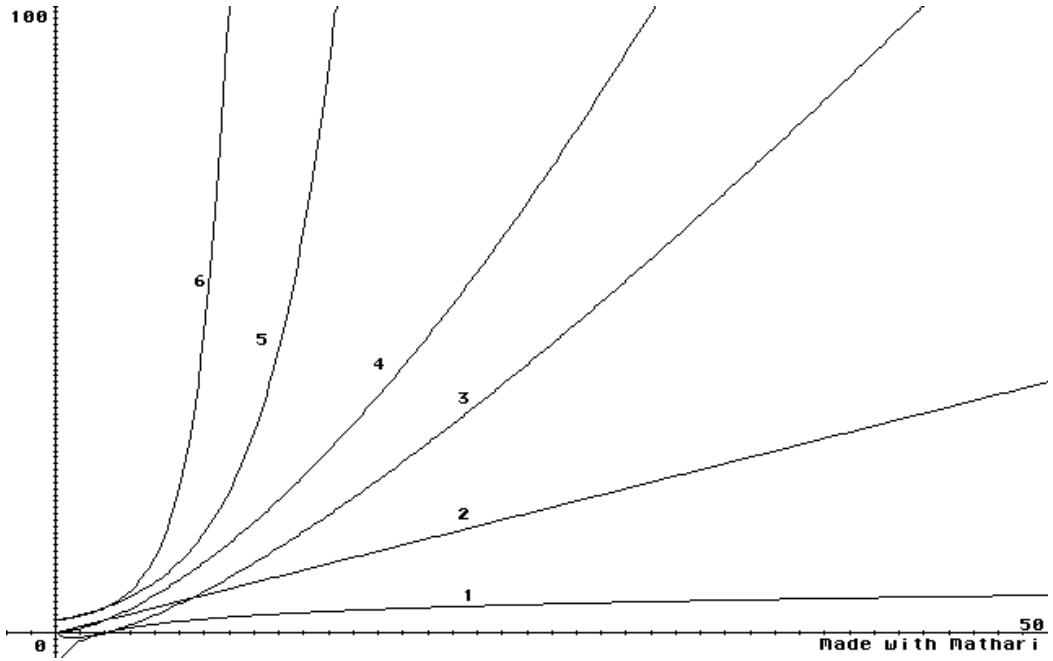
- $T(n) = (n+1)^2$ est en $O(n^2)$
- $Q(n) = 3n + 15$ est en ?
- $R(n) = n^2 + n + 250$ est en ?
- $S(n) = log_2(n) + 25$ est en ?
- $U(n) = 2^n + n^3$ est en ?

Classes de complexité

Plusieurs grandes classes de complexité

- les algorithmes sub-linéaires (temps logarithmique)
  - complexité en $O(log(n))$
  - les plus rapides
- les algorithmes linéaires
  - complexité en $O(n)$ et en $O(n\ log(n))$
  - des algorithmes rapides
- les algorithmes quadratiques
  - complexité en $O(n^2)$
  - des algorithmes avec des performances moyennes
- les algorithmes polynomiaux :
  - complexité en $O(n^k)$ (avec $k > 2$)
  - des algorithmes lents
- les algorithmes exponentiels :
  - complexité supérieur à tout polynôme en n
  - des algorithmes intraitables lorsque la taille des données est supérieur à quelques dizaines d'unités

Classes de complexité



1. complexité logarithmique en $O(ln(n))$ (ou $ln^k(n)$, $k > 1$)
2. complexité linéaire en $O(n)$
3. complexité quasi-linéaire en $O(n\ ln(n))$
4. complexité polynomiale en $O(n^k)$ $(k > 1)$
5. complexité exponentielle en $O(a^n)$ $(a > 1)$
6. complexité hyperexponentielle comme $O(n!)$ ou pire, $O(n^n)$

Comparaison des ordres de grandeur

$O(1) \subset O(log(n)) \subset O(ln(n)) \subset O(n) \subset O(n\ log(n)) \subset O(nln(n)) \subset O(n^k) \subset O(2^n) \subset O(n!) \subset O(n^n)$, avec $k \geq 2$

	2	2^4	2^6	2^8
log(log(n))	0	2	2.58	3
log(n)	1	4	6	8
n	2	16	64	256
n log(n)	2	64	384	2 048
n^2	4	256	4 096	65 536
2^n	4	65 536	1.84e+19	1.15e+77
n !	2	2.09e+13	1.26e+89	8.57e+506

Exemples de complexités et de tâches associées

Complexité	*Tâche*
O(1)	accès direct à un élément
O(log(n))	divisions successives par deux d'un ensemble
O(n)	parcours d'un ensemble
O(n log(n))	divisions successives par deux d'un ensemble et parcours de toutes les parties
O(n^2)	parcours d'une matrice carrée de taille n
O(2^n)	génération des parties d'un ensembles
O(n !)	génération des permutations d'un ensemble

Exercice complexité : égalité de deux matrices

Donner la complexité en temps et en espace de l'algorithme suivant vérifiant l'égalité entre deux matrices carrées de même taille

Fonction EGAL(A,B,l)

Entrée: A et B deux matrices carrées de même taille l
Sortie: vrai si A et B sont égaux, faux sinon

```
1: Pour i de 0 à l − 1 faire
2:     Pour j de 0 à l − 1 faire
3:         Si A[i][j] ≠ B[i][j] Alors
4:             Retourner Faux
5:         Fin Si
6:     Fin Pour
7: Fin Pour
8: Retourner Vrai
```

Calculer la complexité en temps d'un algorithme itératif

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$

une affectation, une opération arithmétique, un accès à une case d'un tableau, etc	1 opération élémentaire
un test d'égalité, une comparaison (pour les types primitifs)	1 opération élémentaire
une boucle `Pour X de i à j` avec $X \leq j$	3 opérations élémentaires $\times\ (j - i + 1)$ itérations
une boucle `Tant que ... faire`	nb d'op. élém. du test $\times$ nb itérations + nb d'op. élém. du test d'arrêt
à l'intérieure d'une boucle	multiplier chaque opérations par le nombre d'itérations de la boucle

- Si le nombre d'itérations d'une boucle dépend des données, introduire une nouvelle variable et étudier la complexité au pire et/ou au meilleur et/ou en moyenne

Calculer la complexité en temps d'un algorithme itératif

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

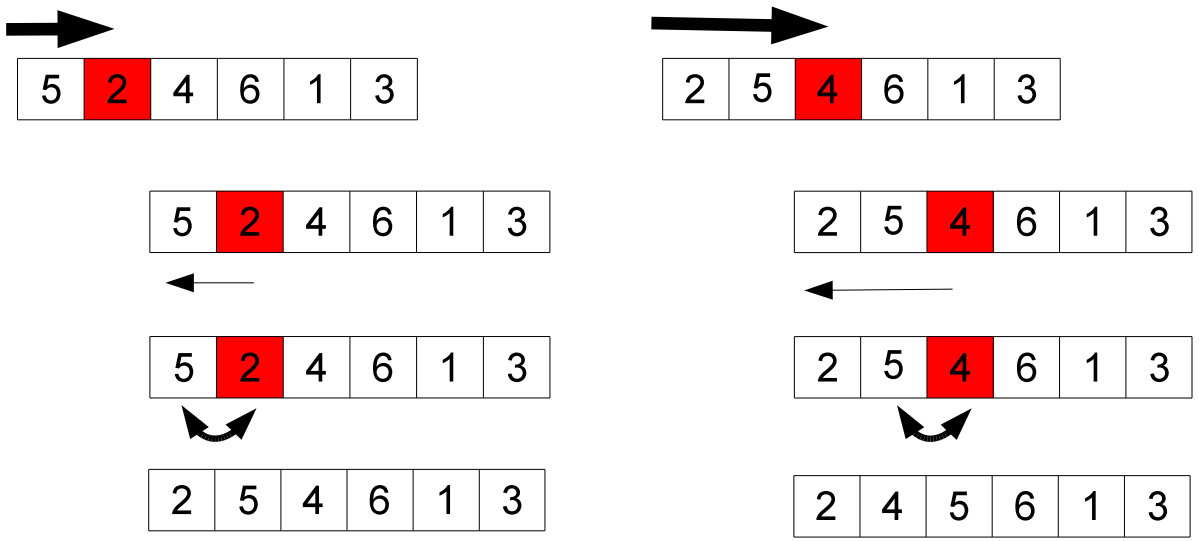
1. Choisir la notation de Landau en fonction de l'approximation souhaitée : O (borne sup.), Ω (borne inf.) et Θ (équivalents)
2. Deviner $g(n)$ à partir des termes d'ordre supérieur
3. Démontrer que $f(n)$ vérifie la notation de Landau en trouvant un n_o et un c

Exemple du tri par insertion

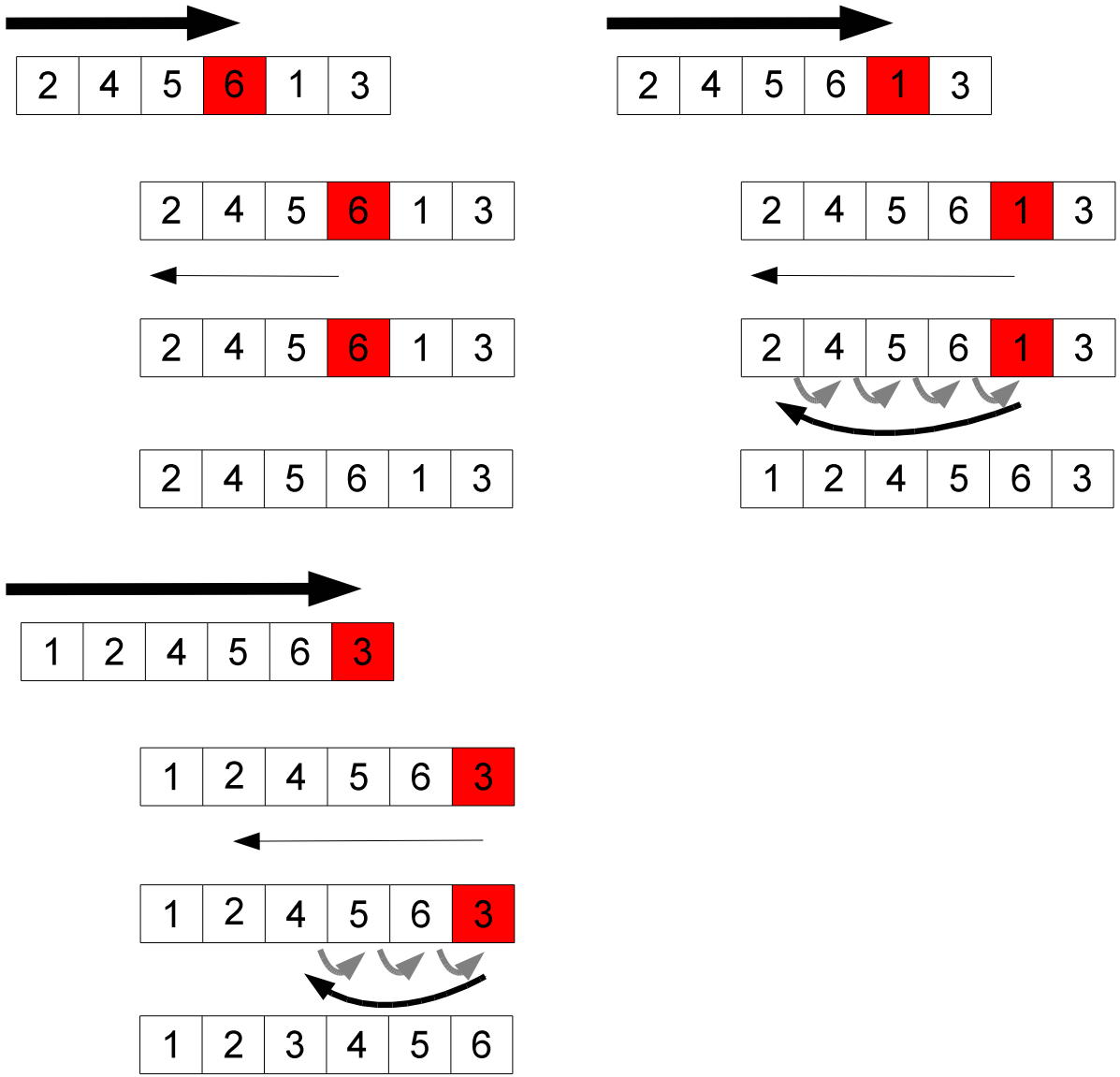
Principe :

- de manière répétée, on choisit un nombre de la séquence d'entrée et on le déplace à la bonne position dans la séquence des nombres déjà triés
- même principe que celui utilisé pour trier une poignée de cartes

Exemple :



Exemple du tri par insertion



Exemple du tri par insertion

Procedure TriInsertion(element[] A)

Entrée: A un tableau d'entier
Sortie: le tableau A trié dans l'ordre croissant

		Nombre d'opérations élémentaires
		Soit n la longueur du tableau A
1:	**Pour** j de 1 à $longueur(A) - 1$ **faire**	3 (n-1) (si longueur(A) $\rightarrow$ 0 op.)
2:	clé$\leftarrow A[j]$	2 (n-1)
3:	$i \leftarrow j - 1$	2 (n-1)
4:	**Tant que** $i >= 0$ et $A[i] >$clé **faire**	$\sum_{j=2}^{n}(3t_j + 3) = 3(n-1) + \sum_{j=2}^{n} 3t_j$
5:	$A[i+1] \leftarrow A[i]$	$\sum_{j=2}^{n} 4t_j$
6:	$i \leftarrow i - 1$	$\sum_{j=2}^{n} 2t_j$
7:	**Fin Tant que**	
8:	$A[i+1] \leftarrow$ clé	3(n-1)
9:	**Fin Pour**	

t_j : nombre d'itérations de la boucle **Tant que** pour une valeur de j donnée $(j \in [2..n])$

Exemple du tri par insertion

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$

- Nombre d'opérations élémentaires du tri pas insertion : $f(n) = 13n - 13 + \sum_{j=2}^{n} 9t_j$, avec valeur de t_j dépendant du contenu du tableau

- Complexité (en temps) au pire :
  - tableau trié dans l'ordre inverse $\rightarrow t_j = j - 1$

  ⇨ $f(n) = \frac{9}{2}n^2 + \frac{17}{2}n - 13$

  $f(n) = 13n - 13 + 9\sum_{j=2}^{n}(j-1)$

  rappel : $\sum_{k=1}^{m} k = \frac{m(m+1)}{2}$

  or $\sum_{k=1}^{m} k = \sum_{k=2}^{m+1}(k-1) = m + \sum_{k=2}^{m}(k-1)$

  donc $\sum_{k=2}^{m}(k-1) = \frac{m(m-1)}{2}$

  $f(n) = 13n - 13 + 9(\frac{n(n-1)}{2})$

Exemple du tri par insertion

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$ (Suite)

- Complexité (en temps) en moyenne :
  - si les nombres sont choisis au hasard, quelle sera la valeur de t_j ? , i.e. où devra-t-on insérer $A[j]$ dans le sous-tableau $A[1..j-1]$?
  - ⇨ $t_j = j/2$ en moyenne
  - ⇨ $f(n) = \frac{9}{4}n^2 + \frac{43}{4}n - 13$

$$f(n) = 13n - 13 + 9(\frac{n(n-1)}{4})$$

- Complexité (en temps) au meilleur :
  - tableau déjà trié $\rightarrow t_j = 0$
  - ⇨ $f(n) = 13n - 13$

Exemple du tri par insertion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

- Complexité (en temps) au pire :
  1. approximation souhaitée : O
     - rappel : $f = O(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times g(n)$
  2. supposons que $g(n) = n^2$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 13$, alors $\forall n \geq n_0, \frac{9}{2}n^2 + \frac{17}{2}n - 13 \leq 13 \times n^2$

  ⇨ l'algorithme est en $O(n^2)$ (i.e. quadratique) dans le pire des cas

- Complexité (en temps) en moyenne :
  1. approximation souhaitée : O
  2. supposons que $g(n) = n^2$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 13$, alors $\forall n \geq n_0, \frac{9}{4}n^2 + \frac{43}{4}n - 13 \leq 13 \times n^2$

  ⇨ l'algorithme est en $O(n^2)$ (i.e. quadratique) en moyenne

Exemple du tri par insertion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

- Complexité (en temps) au meilleur :
  1. approximation souhaitée : O
  2. supposons que $g(n) = n$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 13$, alors $\forall n \geq n_0, 13n - 13 \leq 13 \times n$
  - ⇨ l'algorithme est en $O(n)$ (i.e. linéaire) dans le meilleur cas

Exercice : donner une approximation en Θ de la complexité (en temps) de l'algorithme de tri par insertion.

Calculer la complexité en temps des algorithmes récursifs

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence

- Pourquoi ?
  - impossible de calculer directement le nombre d'opérations à cause des appels récursifs
- Comment ?
  - principe de la récursivité : problème initial divisé en a sous-problèmes chacun de taille $1/b$ de la taille du problème initial

⇨

$$f(n) = \begin{cases} \Theta(1), & \text{si } n \leq c \\ af(n/b) + D(n) + C(n), & \text{sinon} \end{cases}$$

- $af(n/b)$: temps de résolution des a sous-problèmes (avec $n/b = \lfloor n/b \rfloor$ ou $\lceil n/b \rceil$)
- $D(n)$: temps nécessaire à la division du problème en sous-problèmes
- $C(n)$: temps pour construire la solution finale à partir des solutions des sous-problèmes

Calculer la complexité en temps des algorithmes récursifs

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence (Suite)

- Exemple :

Fonction Fact(n)	**Nombre d'opérations élémentaires**
Entrée: un entier $n \geq 0$	*Soit f(n) le nombre d'opérations pour calculer Fact(n)*
Sortie: l'entier $n!$	
1: **Si** n = 0 **Alors**	1
2: **Retourner** 1	1
3: **Fin Si**	
4: res ← Fact(n-1) . n	1 + f(n-1) + 2
5: **Retourner** res	1

- $f(n) = \begin{cases} 2, & \text{si } n \leq 0 \\ f(n-1) + 5, & \text{sinon} \end{cases}$
  - décomposition en $a = 1$ sous-problème de taille $n - 1$
  - temps nécessaire à la division du problème en sous-problèmes : $D(n) = 1$
  - temps pour construire la solution finale à partir des solutions des sous-problèmes : $C(n) = 3$

Calculer la complexité en temps des algorithmes récursifs

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

1. Choisir la notation de Landau en fonction de l'approximation souhaitée : O (borne sup.), Ω (borne inf.) et Θ (équivalents)
2. Deviner $g(n)$
   - attention : impossible d'utiliser les termes d'ordre supérieur
   - ⇨ s'inspirer des récurrences similaires, et dont la solution est connue
   - ⇨ trouver une borne supérieure (et/ou inférieure) très large, puis réduire
     - p.ex. étudier $O(n^2)$, puis $O(n\ log_2(n))$, ...
3. Démontrer que $f(n)$ vérifie la notation de Landau en faisant une démonstration par récurrence
   - 3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0
   - 3.2 faire l'hypothèse que $f(n/b)$ vérifie la notation de Landau
   - 3.3 démontrer que $f(n)$ vérifie la notation de Landau en substituant $f(n/b)$ (cf hypothèse)

Calculer la complexité en temps des algorithmes récursifs

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

Exemple avec l'algorithme récursif $Fact(n)$:

1. approximation souhaitée : O
2. supposons que $g(n) = n$
3. démontrer par récurrence que $f(n) = f(n-1) + 5$ est en $O(n)$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n$
   - 3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0, cad $\exists c > 0, f(n_0) \leq c \times n_0$
     - oui, si $n_0 = 1$ alors $f(1) = f(0) + 5 = 7$ et $f(1) \leq c \times 1, \forall c \geq 7$
   - 3.2 hypothèse de récurrence : $f(n-1)$ en $O(n-1)$, cad $\exists c > 0, f(n-1) \leq c \times (n-1)$
   - 3.3 démontrer que $f(n)$ en $O(n)$
     - $f(n) = f(n-1) + 5$
     - donc $f(n) \leq c \times (n-1) + 5$
     - $f(n) \leq c \times n - c + 5 \leq c \times n$, si $c \geq 5$
     - CQFD

Exemple du tri fusion

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence

Procedure mergeSort(element[] tab, entier i, entier j)

Entrée: une partition du tableau tab située entre les indices i et j
Sortie: la partition triée

```
1: Si i < j Alors
2:     mergeSort(tab, i, (i+j)/2 )
3:     mergeSort( tab, (i+j)/2+1, j)
4:     fusion( tab, i, (i+j)/2, j )
5: Fin Si
```

Nombre d'opérations élémentaires
Soit n la longueur du tableau T et f(n) la fonction représentant le nombre d'opérations

```
1
f(n/2) + 2
f(n/2) + 3
2 + fusion(n) ?
```

⇨ $f(n) = 2f(n/2) + 8 + fusion(n)$

- mais que vaut $fusion(n)$?

Exemple du tri fusion

	Nombre d'opérations élémentaires (dans le pire cas) ***Soit n la longueur du tableau T entre les cases*** deb **et** fin
Procedure fusion(element[] T, entier deb, entier mid, entier fin)	
Entrée: T le tableau initial triés entre deb et mid, et entre mid+1 et fin	
Sortie: le tableau T trié entre deb et fin	
$i \leftarrow 0$; $i_1 \leftarrow deb$; $i_2 \leftarrow mid + 1$	**4**
Tant que $i_1 \leq mid$ **et** $i_2 \leq fin$ **faire**	**2 × (n-1) + 2**
Si $T[i_1] < T[i_2]$ **Alors**	**3 × (n-1)**
$temp[i] \leftarrow T[i_1]$	**3 × (n-1)**
$i_1 \leftarrow i_1 + 1$	**2 × (n-1)**
Sinon	
$temp[i] \leftarrow T[i_2]$	
$i_2 \leftarrow i_2 + 1$	
Fin Si	
$i \leftarrow i + 1$	**2 × (n-1)**
Fin Tant que	
Si $i_1 < mid + 1$ **Alors**	**2**
Pour j **de** i_1 **à** mid **faire**	**3 × 1 (car 1 itération dans le pire des cas)**
$temp[i] \leftarrow T[j]$	**3**
$i \leftarrow i + 1$	**2**
Fin Pour	
Sinon Si $i_2 < fin + 1$ **Alors**	
Pour j **de** i_2 **à** fin **faire**	
$temp[i] \leftarrow T[j]$	
$i \leftarrow i + 1$	
Fin Pour	
Fin Si	
$k \leftarrow 0$	**1**
Pour i **de** deb **à** fin **faire**	**3 × n**
$T[i] \leftarrow temp[k]$; $k \leftarrow k + 1$	**5 × n**
Fin Pour	

Exemple du tri fusion

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence (Suite)

- Dans le pire cas, $fusion(n) = 20n + 5$
- Donc $f(n) = 2f(n/2) + 20n + 13$ avec $f(1) = 1$

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

1. approximation souhaitée : O
2. supposons que $g(n) = n\ log_2(n)$
3. démontrer par récurrence que $f(n) = 2f(n/2) + 20n + 13$ est en $O(n\ log_2(n))$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n \times log_2(n)$

Exemple du tri fusion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

3 démontrer par récurrence que $f(n) = 2f(n/2) + 20n + 13$ est en $O(n\ log_2(n))$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n \times log_2(n)$

3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0, cad $\exists c > 0, f(n_0) \leq c \times n_0 \times log_2(n_0)$

- oui, si $n_0 = 2$ alors $f(2) = 2f(1) + 20 \times 2 + 13 = 55$ et $f(2) \leq c \times 2\log_2(2)$ $(\log_2(2) = 1)$, $\forall c \geq 28$

3.2 hypothèse de récurrence : $f(n/2)$ en $O(\ n/2\ log_2(n/2)\ )$, cad $\exists c > 0, f(n/2) \leq c \times n/2 \times log_2(n/2)$

3.3 démontrer que $f(n)$ en $O(n\ log_2(n))$

- $f(n) = 2f(n/2) + 20n + 13$
- donc $f(n) \leq 2 \times (\ c \times n/2 \times log_2(n/2)\ ) + 20n + 13$
  $\leq c \times n \times (\ log_2(n) - log_2(2)\ ) + 20n + 13$
  $\leq c \times n \times log_2(n) - c \times n + 20n + 13$
- vrai, si $c \times n \geq 20n + 13$, cad si $c \geq 33$ $(n \geq 1)$
- CQFD

Exercices : démonstration par récurrence

En supposant que $f(1) = 1$,

1. approximer (en Θ) $f(n) = 2f(\lfloor n/2 \rfloor) + n$

2. montrer que $f(n) = f(\lfloor n/2 \rfloor) + 1$ est en $O(log_2(n))$

3. montrer que $f(n) = 2f(\lfloor n/2 \rfloor + 2) + n$ est en $O(n\ log_2(n))$
   - avec $f(n) = 1$, tout $0 < n < 5$
   - indication : démontrer d'abord que $f(n) \leq c(n-4)log_2(n-4) - n$ (car $c(n-4)log_2(n-4) - n \leq c \times n \times log_2(n)$, $c > 0$ et $n > 0$)

Pour résumer

Analyse de la complexité des algorithmes

➪ Objectif : choisir le meilleur algorithme en fonction de ses besoins

- en général, un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur
- compromis espace-temps

Techniques :

- complexité en temps :
  - complexité des algorithmes itératifs
    - compter les opérations élémentaires et exprimer la complexité sous la forme d'un fonction dépendant de la taille de l'entrée (n)
    - estimer l'ordre de grandeur (O, Θ) en fonction de la taille de l'entrée
    - trouver des valeurs de n_0 et de c vérifiant la définition
  - complexité des algorithmes récursifs
    - compter les opérations élémentaires et exprimer la complexité sous la forme d'une équation de récurrence
    - estimer l'ordre de grandeur (O, Θ) en fonction de la taille de l'entrée
    - démontrer par récurrence que l'ordre de grandeur est vérifié
- complexité en espace
  - étudier la taille (en mémoire généralement) des structures de données et variables utilisées

Rq : si nécessaire, étudier le pire cas et/ou le cas moyen

Exercice problème de puissance

Donner la complexité (dans le pire cas) en temps et en espace des trois algorithmes suivants permettant de calculer x^k

Fonction Puissance1(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: res ← 1
2: Pour i de 0 à n − 1 faire
3:     res ← res . x
4: Fin Pour
5: Retourner res
```

Fonction Puissance2(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 0 Alors
2:     Retourner 1
3: Sinon
4:     Retourner x.Puissance2( x, n-1 )
5: Fin Si
```

Exercice problème de puissance

Fonction Puissance3(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 0 Alors
2:     Retourner 1
3: Sinon
4:     Si estPair( n ) Alors
5:         Retourner Puissance3( x.x, n/2 )
6:     Sinon
7:         Retourner x.Puissance3( x.x, (n-1)/2 )
8:     Fin Si
9: Fin Si
```