



# LIFAP3 – Algorithmique et programmation avancée

Samir Akkouché

Nicolas Pronost



# Chapitre 4

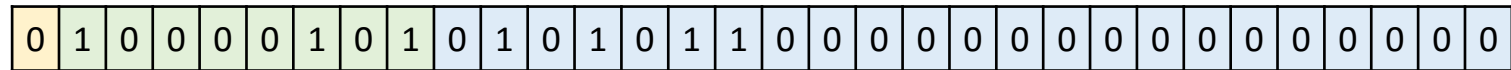
## Fichier, tri et complexité

# Différents types de fichiers

Format des informations	
« texte » (codes de caractères)	« binaire » (comme en mémoire)
Fichiers .cpp	Fichiers objet (.o)
Fichiers Makefile	Fichiers mp3, avi
Fichiers de configuration Linux	Fichiers pdf
Fichiers .xml, .html, .txt	Fichiers exécutables
Etc.	Etc.

# Fichier « texte »

- Les octets du fichier ne contiennent que des codes de caractères
- Les données numériques doivent être converties en caractères
- Exemple: un nombre au format IEEE 754 simple précision



signe    exposant décalé

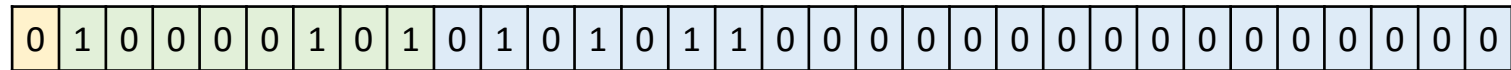
pseudo-mantisse

- Décodage:  $nombre = 1.3359375 \times 2^6 = 85.5 = 8.55e1$
- Ecriture dans le fichier (virgule fixe, 3 chiffres après la virgule) des caractères: '8' '5' '.' '5' '0' '0'

00111000	00110101	00101110	00110101	00110000	00110000
(56='8')	(53='5')	(46='.' )	(53='5')	(48='0')	(48='0')

# Fichier « texte »

- Les octets du fichier ne contiennent que des codes de caractères
- Les données numériques doivent être converties en caractères
- Exemple: un nombre au format IEEE 754 simple précision



signe    exposant décalé

pseudo-mantisse

- Décodage:  $nombre = 1.3359375 \times 2^6 = 85.5 = 8.55e1$
- Ecriture dans le fichier (virgule flottante, 3 chiffres pour la mantisse)  
des caractères: '8' '.' '5' '5' 'e' '1'

00111000

(56='8')

00101110

(53='.'')

00110101

(46='5')

00110101

(53='5')

01100101

(101='e')

00110001

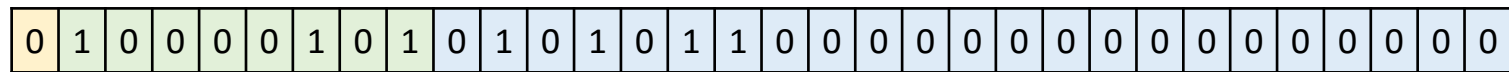
(49='1')

# Fichier « texte »

- Avantages
  - portabilité
  - fichier lisible (et facilement modifiable) avec un simple éditeur de texte
- Inconvénients
  - coût en temps induit par les conversions nécessaires
  - penser à écrire suffisamment de décimales pour ne pas perdre en précision numérique

# Fichier « binaire »

- On recopie l'information comme elle figure en mémoire
- Exemple



signe    exposant décalé

pseudo-mantisse

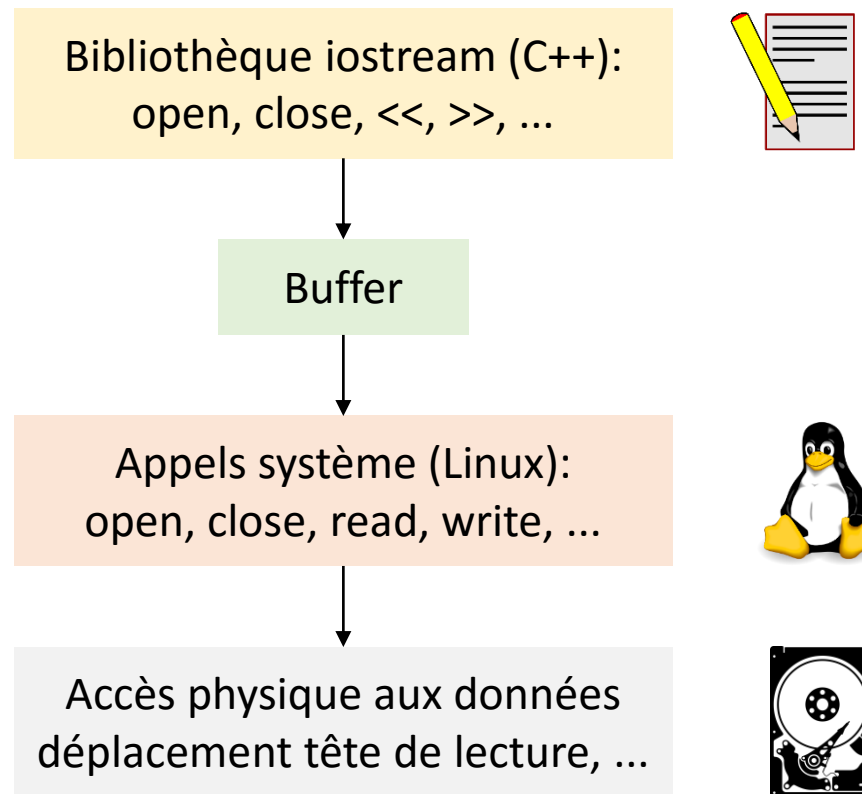
- Ecriture dans le fichier (regroupement en octet juste pour lisibilité)

01000010   10101011   00000000   00000000

- Avantages
  - fichier plus compact (n'utilise pas de codage intermédiaire)
  - lecture/écriture sans conversion = manipulation plus rapide
  - pas de formatage de présentation = pas de perte de précision
- Inconvénients
  - fichier illisible (et difficilement éditable) par un éditeur de texte
  - il faut gérer les problèmes de portabilité
  - il faut savoir qu'est ce qui est écrit pour pouvoir le relire

# Utilisation depuis un programme

- Doit-on se soucier du support physique des fichiers?
- Exemple en C++ sous Linux:





# Notion de buffer

- Le buffer est une structure de donnée de type FIFO
  - First in, first out
  - File (*cf. cours sur les files plus tard*)
- Espace de stockage temporaire pour limiter le nombre d'appels système qui sont lents (l'accès physique au disque dur est l'une des opérations les plus lente sur un ordinateur)
- Un curseur indique où on en est dans la lecture/écriture

# Les fichiers en C++

- C++ fournit trois classes pour manipuler des fichiers
  - **ofstream** est la classe pour écrire dans un fichier (o pour out)
  - **ifstream** est la classe pour lire depuis un fichier (i pour in)
  - **fstream** est la classe pour lire et écrire
- Ces classes fonctionnent exactement comme les classes **ostream** et **istream** déjà utilisées pour afficher à l'écran et saisir au clavier
- On utilise les mêmes opérateurs >> et <<
  - Ces opérateurs sont surchargés dans les classes **fstream**
- On a juste besoin de spécifier le fichier physique dans lequel lire/écrire

# Ouvrir un fichier

- Pour associer un fichier physique à un flux dans lequel on lit ou écrit, il faut l'ouvrir avec la fonction membre **open**

```
void open (const char *nom_fichier, openmode mode);
```

- nom\_fichier est la chaîne de caractères représentant le nom du fichier physique
  - relatif au dossier dans lequel le programme est exécuté (donc inclure des changements de dossier si le fichier n'est pas dans le même)
- mode est un paramètre optionnel qui indique le mode d'ouverture
  - parmi les valeurs possibles de mode, on trouve les suivantes :

<code>ios::in</code>	Ouverture en lecture
<code>ios::out</code>	Ouverture en écriture
<code>ios::app</code>	Ouverture en mode ajout en fin (append)

# Ouvrir un fichier

- Les modes d'ouverture peuvent être combinés en utilisant l'opérateur OU (|)
- Chaque fonction membre **open** des classes **fstream**, **ofstream** et **ifstream** a un mode d'ouverture par défaut qu'il est inutile de préciser si seul le nom du fichier est donné en paramètre

ofstream	ios::out
ifstream	ios::in
fstream	ios::in   ios::out

# Ouvrir un fichier

- Exemples
  - Pour ouvrir un fichier en lecture

```
ifstream monFichier;  
monFichier.open("fichier.txt");
```

- Pour ouvrir un fichier en écriture

```
ofstream monFichier;  
monFichier.open("fichier.txt");
```

- Pour ouvrir un fichier en écriture en mode ajout (*append*)

```
ofstream monFichier;  
monFichier.open("fichier.txt", ios::app);
```

# Ouvrir un fichier

- Comme on souhaite très souvent créer un flux et ouvrir un fichier en même temps, les trois classes ont un constructeur qui prend en paramètre les mêmes paramètres que la fonction membre **open**
- Les exemples précédents peuvent donc aussi s'écrire :

```
ifstream monFichier ("fichier.txt");
```

```
ofstream monFichier ("fichier.txt");
```

```
ofstream monFichier ("fichier.txt", ios::app);
```

# Ouvrir un fichier

- L'opération d'ouverture n'est pas forcément réussie
  - Ex. un fichier en lecture qui n'existe pas
  - Ex. un fichier en lecture ou écriture dont vous n'avez pas les droits
  - Ex. un fichier en écriture alors qu'il n'y a plus de place sur le DD
- La fonction membre **is\_open** retourne une valeur booléenne indiquant si l'ouverture a été un succès



Il faut absolument, toujours, vérifier si l'ouverture a été un succès

- manipuler un flux non ouvert produit un crash à l'exécution

```
ofstream monFichier ("fichier.txt");  
if (monFichier.is_open()) { /* OK, on peut travailler sur le flux */ }  
else { /* Attention, erreur dans l'ouverture */ }
```

# Fermeture d'un fichier

- Quand les opérations d'écriture et de lecture sont finies, il ne faut pas oublier de fermer le fichier

```
monFichier.close();
```

- Afin de notifier l'OS qu'il peut libérer la ressource et la rendre disponible à nouveau
- En effet, deux programmes (ou deux blocs d'instructions d'un même programme) n'ont pas le droit d'avoir le même fichier ouvert en même temps
  - pour des raisons évidentes de conflits
- Une fois la fermeture effectuée, le flux peut être réutilisé pour ouvrir un autre fichier, et le fichier peut être ré-ouvert par un autre flux
- Si un objet contenant un flux est détruit alors que le flux est encore ouvert, le destructeur de l'objet appelle automatiquement la fermeture du fichier



# Ecriture dans un fichier

- Les écritures dans le flux fichier se font de la même manière que l'affichage sur l'écran: avec **l'opérateur <<**

```
ofstream monFichier ("fichier.txt");
Date maDate(1,2,2000);
if (monFichier.is_open()) {
    monFichier << "La somme de " << 10 << " et " << 5 << " vaut " << 10+5;
    monFichier << endl << " en date du " << maDate;
    monFichier.close();
}
else {
    cout << "Erreur d'ouverture du fichier: " << "fichier.txt" ;
}
```

produit le fichier physique fichier.txt avec le contenu:

```
La somme de 10 et 5 vaut 15
en date du 1/2/2000
```

# Lecture depuis un fichier

- Les lectures dans le flux fichier peuvent se faire de la même manière que la saisie au clavier: avec **l'opérateur >>**
- Mais il extrait les données une à une en prenant en compte les caractères de séparation (espace, tabulation, fin de ligne etc.)
  - il permet de faire une extraction **mot à mot**
- De plus il ne fait pas la distinction entre caractères provenant d'un mot et nombres convertis en caractères lors de l'écriture
  - Tout est caractère pour cet opérateur

# Lecture depuis un fichier

- Donc ici il faudrait le code suivant

```
ifstream monFichier ("fichier.txt");
string mot;
Date maDate;
int x,y,z;
if (monFichier.is_open()) {
    monFichier >> mot >> mot >> mot ;           // La somme de (3 mots)
    monFichier >> x >> mot >> y >> mot >> z;      // x et y vaut z (2 mots)
    monFichier >> mot >> mot >> mot >> maDate;    // en date du maDate (3 mots)
    monFichier.close();
}
else {
    cout << "Erreur d'ouverture du fichier: " << "fichier.txt" ;
}
```

pour affecter les valeurs

```
x = 10 , y = 5 , z = 15 et date = (1,2,2000)
```

# Lecture depuis un fichier

- Pour lire une ligne entière dans le fichier, on peut utiliser la fonction **getline**
  - depuis le curseur courant jusqu'au prochain caractère de fin de ligne
- La chaîne de caractères résultant peut ensuite être traitée pour récupérer des données, rechercher des sous chaînes...
  - toutes les fonctionnalités de la classe string sont alors disponibles

```
ifstream monFichier ("fichier.txt");
string ligne;
if (monFichier.is_open()) {
    getline(monFichier,ligne);
    // traitement de la première ligne (la somme de x et y vaut z)
    getline(monFichier,ligne);
    // traitement de la deuxième ligne ( en date du maDate)
    monFichier.close();
}
else {
    cout << "Erreur d'ouverture du fichier: " << "fichier.txt" ;
}
```

# Contrôle de l'état du flux

- Si vous ne connaissez pas à l'avance le nombre de caractères/lignes du fichier, vous pouvez tester si le curseur est en fin de fichier avec la fonction membre **eof**

```
ifstream monFichier ("fichier.txt");
if (monFichier.is_open()) {
    while (!monFichier.eof()) {
        // lecture contenu:
        // ex. monFichier >> mot; ou bien getline(monFichier,ligne);
    }
    // traitement de fin de fichier
    monFichier.close();
}
else {
    cout << "Erreur d'ouverture du fichier: " << "fichier.txt" ;
}
```

# Contrôle dans le flux

- En interne, le fichier est stocké dans un buffer (pour accélérer les accès)
- On a la possibilité (rarement utile) de manipuler le curseur indiquant où on en est dans la lecture/écriture du buffer
  - On peut donc « sauter/relire/réécrire » des données déjà lues/écrites
- Les fonctions membres **tellg** et **tellp** permettent de récupérer les positions des curseurs de lecture et d'écriture
- Les fonctions membres **seekg** et **seekp** permettent de modifier la position de ces curseurs

# Contrôle dans le flux

- Exemple pour calculer la taille d'un fichier

```
streampos debut,fin;
//streampos est le type des positions des curseurs

ifstream monFichier ("fichier.txt");
// ouverture d'un fichier en lecture

debut = monFichier.tellg();
// position du curseur après ouverture (début)

monFichier.seekg(0, ios::end);
// se déplace à la fin (décalage de 0 par la rapport à la fin)

fin = monFichier.tellg();
// position du curseur (fin)

monFichier.close();
// fermeture du fichier

cout << "la taille du fichier est: " << (fin-debut) << " octets";
// on peut effectuer des opérations sur des streampos (equiv. int)
```

# Mode binaire

- En mode binaire, on n'a pas besoin de formater les données du fichier (pas des caractères, pas de retour à la ligne, ...)
- On ne doit pas utiliser les opérateurs >> et <<, mais les fonctions membres d'un flux **write** et **read**
  - pour écrire les n premiers caractères du tableau s dans le flux

```
ostream& write (const char* s, streamsize n)
```

- pour lire n caractères depuis le flux et les stocker dans le tableau s

```
istream& read (char* s, streamsize n)
```



# Mode binaire

- Exemple pour recopier un fichier binaire

```
ifstream monFichier ("fichier.bin"); // ouverture fichier à copier
ofstream copieFichier ("copie.bin"); // ouverture fichier de copie

monFichier.seekg (0, monFichier.end); // placement en fin de fichier
streampos size = monFichier.tellg(); // calcul taille fichier en octets
monFichier.seekg (0);                // placement en début de fichier

char* buffer = new char[size];      // allocation des octets nécessaires à la copie

monFichier.read (buffer,size);       // lecture du contenu entier en une lecture

copieFichier.write (buffer,size);    // écriture du contenu entier en une écriture

delete[] buffer;                    // libération de la mémoire allouée

copieFichier.close();                // fermeture des fichiers
monFichier.close();
```

# Mode binaire

- Exemple pour écrire puis lire les nombres de 0 à 10

```
ofstream monFichier ("fichier.txt");  
for (int i=0;i<=10;i++)  
    monFichier.write((const char*) (&i), sizeof(i));  
}  
monFichier.close();
```

```
ifstream monFichier ("fichier.txt");  
for (int i=0;i<=10;i++)  
    int val;  
    monFichier.read((char*) (&val), sizeof(i));  
}  
monFichier.close();
```

# Analyse des performances

- De quoi dépend le temps d'exécution d'un programme?
  - du nombre d'opérations à réaliser, donc du nombre d'objets à traiter et de la quantité de manipulations à effectuer
  - du langage
  - du compilateur
  - de la machine (processeur, temps d'accès à la mémoire...)
- Une **analyse empirique** peut mesurer le temps d'exécution en testant des données en entrée de différentes tailles et les comparer
- Une **analyse formelle** peut
  - comparer la performance de deux algorithmes au niveau du principe, en ignorant les détails bas niveau comme le langage ou le processeur
  - prédire comment l'algorithme se comportera si on augmente la taille des données en entrée (sans avoir à l'exécuter, évite les problèmes d'interprétation des résultats)

# Analyse formelle de la complexité

- Nous allons calculer la complexité d'un algorithme de recherche du maximum dans un tableau de réels dans le pire des cas
  - c'est le cas intéressant, celui qui assure que la complexité ne sera jamais pire que celle obtenue (jamais de mauvaise surprise lors de l'exécution)

```
Variables
  tab : tableau [1...n] de réels
  i : entier
  max : réel
Début
  max ← tab[1]
  pour i allant de 2 à n par pas de 1 faire
    si (tab[i] > max) alors
      max ← tab[i]
    fin si
  fin pour
  afficher(max)
Fin
```

# Analyse formelle de la complexité

- Etape 0: réécrire les pour en tant-que de l'algorithme afin de faire apparaître les opérations élémentaires

```
max ← tab[1]
i ← 2
tant que i <= n faire
    si (tab[i] > max) alors
        max ← tab[i]
    fin si
    i ← i + 1
fin tant que
```

# Analyse formelle de la complexité

- Etape 1: recenser les opérations élémentaires

```
max ← tab[1]
i ← 2
tant que i ≤ n faire
    si (tab[i] > max) alors
        max ← tab[i]
    fin si
    i ← i + 1
fin tant que
```

Dans cet algorithme, on a:

- Accès à un élément de tableau
- Affectation d'entier
- Affectation de réel
- Comparaison de deux réels
- Comparaison de deux entiers
- Addition de deux entiers

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1						
2						
3						
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2						
3						
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```



# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3						
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3						
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```



Combien de fois teste-t-on la condition du tant que?

- jusqu'à ce qu'elle devienne fausse, c.-à-d. quand  $i=n+1$  inclus, soit de 2 à  $n+1$  inclus =  $n$  comparaisons.

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```



Combien de fois teste-t-on la condition du tant que?

- jusqu'à ce qu'elle devienne fausse, c.-à-d. quand  $i=n+1$  inclus, soit de 2 à  $n+1$  inclus =  $n$  comparaisons.

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4						
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```



Si l'on fait x tests de la condition du tant-que, alors on exécute x-1 fois l'intérieur du tant-que

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```



Si l'on fait x tests de la condition du tant-que, alors on exécute x-1 fois l'intérieur du tant-que

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5						
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

Dans **le pire des cas**, la condition du si est toujours vraie (**tableau trié**), même nombre d'exécutions du corps du si que le test de sa condition

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5	n-1		n-1			
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

Dans **le pire des cas**, la condition du si est toujours vraie (**tableau trié**), même nombre d'exécutions du corps du si que le test de sa condition

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5	n-1		n-1			
6						
7						
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

On ne fait aucune d'opération élémentaire sur les sorties de boucle/fonction



# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5	n-1		n-1			
6						
7		n-1				n-1
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

# Analyse formelle de la complexité

- Etape 2: compter le nombre de fois où chaque opération est effectuée, pour chaque ligne

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5	n-1		n-1			
6						
7		n-1				n-1
8						

```
1  max ← tab[1]
2  i ← 2
3  tant que i <= n faire
4      si (tab[i] > max) alors
5          max ← tab[i]
6      fin si
7      i ← i + 1
8  fin tant que
```

On ne fait aucune d'opération élémentaire sur les sorties de boucle/fonction

# Analyse formelle de la complexité

- Etape 3: faire les totaux par opération élémentaire

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Comp. réel	Add. entier
1	1		1			
2		1				
3				n		
4	n-1				n-1	
5	n-1		n-1			
6						
7		n-1				n-1
8						
Total	2n-1	n	n	n	n-1	n-1

# Analyse formelle de la complexité

- Etape 4: appliquer comme constantes multiplicatives les durées (ou autre mesure comme l’empreinte mémoire) des opérations élémentaires
  - Notons  $\tau_{acc}$  le temps en microsecondes pour effectuer un accès dans un tableau,  $\tau_{affe}$  le temps pour effectuer une affectation d’entier, etc.

Ligne	Accès tableau	Aff. entier	Aff. réel	Comp. entier	Camp. réel	Add. entier
Cstes	$\tau_{acc}$	$\tau_{affe}$	$\tau_{affr}$	$\tau_{cmpe}$	$\tau_{cmpr}$	$\tau_{adde}$
Total #	$2n-1$	$n$	$n$	$n$	$n-1$	$n-1$
Temps	$\tau_{acc}(2n - 1) + \tau_{affe}n + \tau_{affr}n + \tau_{cmpe}n + \tau_{cmpr}(n - 1) + \tau_{adde}(n - 1)$					

# Analyse formelle de la complexité

- Etape 5: réorganiser les termes en fonction du degré de  $n$

$$T(n) = \tau_{acc}(2n - 1) + \tau_{affe}n + \tau_{affr}n + \tau_{cmpe}n + \tau_{cmpr}(n - 1) + \tau_{adde}(n - 1)$$

$$T(n) = n(2\tau_{acc} + \tau_{affe} + \tau_{affr} + \tau_{cmpe} + \tau_{cmpr} + \tau_{adde}) + (-\tau_{acc} - \tau_{cmpr} - \tau_{adde})$$

- Ce temps d'exécution est de la forme  $an + b$ 
  - les constantes  $a$  et  $b$  dépendent du langage, du compilateur, du matériel etc. mais la forme est vraie quelque soit l'environnement utilisé
- On dit que cet algorithme est de complexité **linéaire** dans le pire des cas
  - Si on double la taille du tableau ( $2n$ ), le temps d'exécution va au pire doubler
  - C'est une propriété intrinsèque de l'algorithme de recherche de maximum dans un tableau

# Analyse formelle de la complexité

- L'analyse formelle regarde qu'est ce qui se passe lorsque  $n$  devient très grand: comportement asymptotique

- Notation  $O$  (grand-o)

- Etant donné deux fonctions  $f(n)$  et  $g(n)$ , on dit que  $f(n)$  est  $O(g(n))$  s'il existe une constante positive  $c$  et un entier constant  $n_0 \geq 1$  tels que

$$f(n) \leq c \times g(n) \text{ pour tout } n \geq n_0$$

- Cela signifie que le taux de croissance de  $f$  n'est pas aussi grand que celui de  $g$
  - Le taux de croissance n'est pas affecté par
    - les facteurs constants et les termes d'ordres inférieurs
    - donc supprimer les de  $g(n)$
  - *Plus en LIFAP6*

# Les algorithmes de tri

- En algorithmique il est souvent utile de pouvoir trier un ensemble d'objets pour faire, en autres, des recherches rapides d'éléments
  - nécessite de maintenir à jour l'ensemble trié à chaque ajout ou de trier les objets régulièrement
  - nécessite de donner une relation d'ordre entre les objets
    - implémentée en général par l'opérateur  $<$  (strict. inférieur)
  - les objets sont souvent organisés dans un tableau, le tri consiste alors à déplacer les (pointeurs sur) objets d'un élément à un autre
- Les algorithmes de tri ont des propriétés différentes (ex. complexité)
  - Choisir le bon tri pour la bonne utilisation
  - Tri par sélection, tri par insertion, tri rapide, tri fusion, tri bulle, etc.

# Tri par sélection

- Principe: pour chaque élément  $i$  de 1 à  $n-1$ , échanger  $\text{tab}[i]$  avec l'élément minimum de  $\text{tab}[i...n]$ 
  - rechercher les minimums d'un sous-tableau de plus en plus petit
  - les éléments à gauche de  $i$  sont à leur place définitive (et ne seront plus modifiés), donc le tableau est complètement trié lorsque  $i$  arrive sur l'avant-dernier élément (le dernier élément étant forcément le plus grand)

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7



# Tri par sélection

**Procédure** tri\_sélection (tab: tableau [1...n] de Element)

**Précondition:** les éléments de tab sont initialisés

**Post-condition:** les éléments de tab sont triés (plus petits à gauche)

**Paramètre en mode donnée-résultat:** tab

**Variables locales:**

    i, j, indmin : entiers

    min : Element

**Début**

    pour i allant de 1 à n-1 par pas de 1 faire

        indmin ← i

        pour j allant de i+1 à n par pas de 1 faire

            si tab[j] < tab[indmin] alors

                indmin ← j

        finsi

    finpour

    min ← tab[indmin]

    tab[indmin] ← tab[i]

    tab[i] ← min

finpour

**Fin** tri\_sélection

# Tri par sélection

**Procédure** tri\_sélection (tab: tableau [1...n] de **Element**)

**Précondition:** les éléments de tab sont initialisés

**Post-condition:** les éléments de tab sont triés (plus petits à gauche)

**Paramètre en mode donnée-résultat:** tab

**Variables locales:**

    i, j, indmin : entiers

    min : Element

**Début**

    pour i allant de 1 à n-1 par pas de 1 faire

        indmin ← i

        pour j allant de i+1 à n par pas de 1 faire

            si tab[j] < tab[indmin] alors

                indmin ← j

        finsi

    finpour

    min ← tab[indmin]

    tab[indmin] ← tab[i]

    tab[i] ← min

finpour

**Fin** tri\_sélection

Element désigne le type d'objet à trier, ex. réel, entier, Date, ...

# Tri par sélection

**Procédure** tri\_sélection (tab: tableau [1...n] de Element)

**Précondition:** les éléments de tab sont initialisés

**Post-condition:** les éléments de tab sont triés (plus petits à gauche)

**Paramètre en mode donnée-résultat:** tab

**Variables locales:**

i, j, indmin : entiers

min : Element

**Début**

**pour** i allant de 1 à n-1 par pas de 1 faire

indmin ← i

**pour** j allant de i+1 à n par pas de 1 faire

si tab[j] < tab[indmin] alors

indmin ← j

finsi

finpour

min ← tab[indmin]

tab[indmin] ← tab[i]

tab[i] ← min

finpour

**Fin** tri\_sélection

**Boucle pour placer le bon élément (le min) à l'indice i**

# Tri par sélection

**Procédure** tri\_sélection (tab: tableau [1...n] de Element)

**Précondition:** les éléments de tab sont initialisés

**Post-condition:** les éléments de tab sont triés (plus petits à gauche)

**Paramètre en mode donnée-résultat:** tab

**Variables locales:**

i, j, indmin : entiers

min : Element

**Début**

pour i allant de 1 à n-1 par pas de 1 faire

indmin ← i

pour j allant de i+1 à n par pas de 1 faire

si tab[j] < tab[indmin] alors

indmin ← j

finsi

finpour

min ← tab[indmin]

tab[indmin] ← tab[i]

tab[i] ← min

finpour

**Fin** tri\_sélection

Boucle pour recherche le min dans tab[i+1...n]

# Tri par sélection

**Procédure** tri\_sélection (tab: tableau [1...n] de Element)

**Précondition:** les éléments de tab sont initialisés

**Post-condition:** les éléments de tab sont triés (plus petits à gauche)

**Paramètre en mode donnée-résultat:** tab

**Variables locales:**

i, j, indmin : entiers

min : Element

**Début**

pour i allant de 1 à n-1 par pas de 1 faire

indmin ← i

pour j allant de i+1 à n par pas de 1 faire

si tab[j] < tab[indmin] alors

indmin ← j

finsi

finpour

min ← tab[indmin]

tab[indmin] ← tab[i]

tab[i] ← min

finpour

**Fin** tri\_sélection

Permutation de l'élément i avec le min trouvé

# Invariant de boucle

- L'invariant de boucle est la propriété des données vérifiant les conditions suivantes
  - Initialisation: la propriété est vraie avant la 1<sup>ère</sup> itération
  - Conservation: si la propriété est vraie à l'itération  $i$ , alors elle reste vraie à l'itération  $i+1$
  - Terminaison: une fois la boucle terminée, la propriété est utile pour montrer la validité de l'algorithme

Quel est l'invariant de boucle pour le tri par sélection?

# Tri par sélection: invariant de boucle

- L'invariant de boucle est

Si  $i \geq 2$ , `tab` est trié entre les indices 1 et  $i-1$ , et tous les éléments restants sont supérieurs ou égaux à `tab[i-1]`

- Initialisation
  - Pour  $i=2$  (on vient de faire l'affectation  $i \leftarrow 2$  mais on n'a pas exécuté le corps de la boucle pour  $i=2$ ), a-t-on la propriété suivante: « `tab` est trié entre les indices 1 et 1, et tous les éléments restants sont supérieurs ou égaux à `tab[1]` »?
  - Entre les indices 1 et 1, il n'y a qu'un seul élément donc ce sous-tableau est forcément trié. Par ailleurs, l'élément qui se trouve en position 1 est le min de tout le tableau, donc les éléments restants lui sont donc bien supérieurs ou égaux

# Tri par sélection: invariant de boucle

- L'invariant de boucle est

Si  $i \geq 2$ , `tab` est trié entre les indices 1 et  $i-1$ , et tous les éléments restants sont supérieurs ou égaux à `tab[i-1]`

- Conservation

- Supposons que la propriété soit vraie à l'itération  $i$ , soit « `tab` est trié entre les indices 1 et  $i-1$ , et tous les éléments restants sont supérieurs ou égaux à `tab[i-1]` »
- On doit montrer que la propriété reste vraie pour  $i+1$ , soit « `tab` est trié entre les indices 1 et  $i$ , et tous les éléments restants sont supérieurs ou égaux à `tab[i]` »
- Or, lors de l'itération  $i$ , on prend un élément dans la partie non triée pour le mettre à la place  $i$ , et on ne touche pas aux éléments `tab[1...i-1]`. Cet élément est supérieur ou égal à `tab[i-1]` donc le début du tableau restera bien trié, jusqu'à  $i$  inclus maintenant
- Comme par ailleurs, cet élément est le minimum de la partie non triée, les éléments restants sont bien supérieurs ou égaux à `tab[i]`



# Tri par sélection: invariant de boucle

- L'invariant de boucle est

Si  $i \geq 2$ , `tab` est trié entre les indices 1 et  $i-1$ , et tous les éléments restants sont supérieurs ou égaux à `tab[i-1]`

- Terminaison
  - Pour  $i=n$  (dernière valeur prise par  $i$ , qui va causer la sortie de la boucle), l'invariant de boucle s'écrit « `tab` est trié entre les indices 1 et  $n-1$ , et tous les éléments restants sont supérieurs ou égaux à `tab[n-1]` »
  - Donc on a un tableau trié jusqu'à  $n-1$ , suivi d'une valeur supérieure ou égale à toutes les autres. Le tableau est donc bien trié de 1 à  $n$ , ce qui prouve que l'algorithme de tri est correct.

# Quelques complexités de tris

Nom	Cas optimal	Cas moyen	Pire des cas	Complexité spatiale
Tri rapide	$n \log n$	$n \log n$	$n^2$	$n$
Tri fusion	$n \log n$	$n \log n$	$n \log n$	$n$
Tri par insertion	$n$	$n^2$	$n^2$	1
Tri par sélection	$n^2$	$n^2$	$n^2$	1

Vous allez vérifier ça par l'exemple en TP4

# Quelques complexités de tris

Nom	Cas optimal	Cas moyen	Pire des cas	Complexité spatiale
Tri rapide	$n \log n$	$n \log n$	$n^2$	$n$
Tri fusion	$n \log n$	$n \log n$	$n \log n$	$n$
Tri par insertion	$n$	$n^2$	$n^2$	1
Tri par sélection	$n^2$	$n^2$	$n^2$	1

En effet, même si le tableau est déjà trié, on doit faire une double boucle imbriquée pour placer tous les  $i$ .

# Quelques complexités de tris

Nom	Cas optimal	Cas moyen	Pire des cas	Complexité spatiale
Tri rapide	$n \log n$	$n \log n$	$n^2$	$n$
Tri fusion	$n \log n$	$n \log n$	$n \log n$	$n$
Tri par insertion	$n$	$n^2$	$n^2$	1
Tri par sélection	$n^2$	$n^2$	$n^2$	1

On a en effet besoin que d'un nombre constant de variables locales (i.e. quatre) pour trier le tableau entier, indépendamment de sa taille (il serait plus grand on n'aurait pas besoin de plus de variables). Le tri par sélection est un tri sur place (réorganisation du tableau original directement, sans faire de copie).

# Importance de la complexité

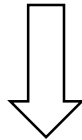
- Exemple: tri d'un tableau de  $n = 10^6$  nombres par ordre croissant



Ordinateur rapide: 1 milliard d'instructions par seconde

Très bon programmeur, langage bas niveau, etc. :  $\tau_{ins} = 2\text{ ms}$

Tri par insertion



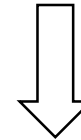
2000 secondes



Ordinateur 100 fois plus lent: 10 millions d'instructions par seconde

Programmeur débutant, langage haut niveau, compilateur non optimisé :  $\tau_{ins} = 50\text{ ms}$

Tri par fusion



100 secondes !

# Tri par insertion

- Principe: les éléments sont insérés les uns après les autres dans un tableau que l'on maintient trié
  - Insérer le premier élément du sous-tableau restant à trier dans la partie déjà triée du tableau
  - Itérer la procédure jusqu'au tri complet du tableau

6 5 3 1 8 7 2 4

- Algorithme, invariant de boucle et complexité en TD
- Implémentation et mesure en TP

# Tri externe

- Parfois, le nombre d'objets à trier est tel qu'on ne peut pas les stocker tous en mémoire
- Exemples
  - articles d'un gros catalogue de vente par correspondance
  - données d'un recensement national
- Il faut des algorithmes de **tri externe**, c'est-à-dire capables de travailler en ne stockant en mémoire que des petites parties du tableau

# Tri externe: caractéristiques

- Pendant le tri, seule une partie des données se trouve en mémoire, le reste est stocké sur disque, bande magnétique, en ligne, etc.
- Accéder aux données est donc beaucoup plus coûteux que de les comparer ou de faire des opérations arithmétiques dessus
- Il peut y avoir des restrictions fortes sur l'accès
  - ex. si accès physique, accès uniquement séquentiel




# Tri externe

- Critères pour les algorithmes
  - Minimiser le nombre de fois où l'on accède à un élément
  - Accéder séquentiellement aux éléments
- Les deux algorithmes de tri externe les plus courants sont
  - le tri fusion (*merge sort*)
  - le tri par paquets (*bucket sort*)

# Notion de monotonie

- Une monotonie est une sous-séquence d'éléments **triés**
- Toute sous-séquence de taille 1 est trivialement une monotonie
- Exemple

56	12	16	23	28	6	2	1	57	59	45
----	----	----	----	----	---	---	---	----	----	----




11 monotonies de longueur 1

# Notion de monotonie

- Une monotonie est une sous-séquence d'éléments **triés**
- Toute sous-séquence de taille 1 est trivialement une monotonie
- Exemple

56	12	16	23	28	6	2	1	57	59	45
----	----	----	----	----	---	---	---	----	----	----



5 monotonies de longueur 2

# Notion de monotonie

- Une monotonie est une sous-séquence d'éléments **triés**
- Toute sous-séquence de taille 1 est trivialement une monotonie
- Exemple

56	12	16	23	28	6	2	1	57	59	45
----	----	----	----	----	---	---	---	----	----	----


The diagram illustrates three monotonic subsequences of length 3 from the array [56, 12, 16, 23, 28, 6, 2, 1, 57, 59, 45]. Red curly braces are used to group the elements: the first brace groups 12, 16, and 23; the second brace groups 16, 23, and 28; and the third brace groups 57, 59, and 45.

3 monotonies de longueur 3

# Notion de monotonie

- Une monotonie est une sous-séquence d'éléments **triés**
- Toute sous-séquence de taille 1 est trivialement une monotonie
- Exemple

56	12	16	23	28	6	2	1	57	59	45
----	----	----	----	----	---	---	---	----	----	----



1 monotonie de longueur 4

# Fusion de deux monotonies

- Fusionner deux monotonies consiste à construire une nouvelle monotonie comprenant tous les éléments des deux monotonies initiales
  - on prend le plus petit des 2 éléments accessibles (ceux en tête de chaque monotonie) et on le recopie dans la nouvelle monotonie
  - on passe à l'élément suivant sur la monotonie dont on a enlevé l'élément de tête
  - on recommence ce processus jusqu'à épuisement d'une monotonie
  - on recopie le reste de la monotonie non encore copié
- A chaque instant, on n'a besoin en mémoire que des deux éléments à comparer
  - les monotonies sont stockées dans un fichier (lecture/écriture quand nécessaire)
- On accède bien séquentiellement aux éléments d'une monotonie donnée

# Principe du tri fusion

- Approche top-down (utilisée en tri interne)
- Approche bottom-up (utilisée en tri externe)
  1. Fusionner les paires d'éléments successifs pour obtenir une séquence de  $n/2$  monotonies de longueur 2
  2. Fusionner les paires de monotonies successives pour obtenir une séquence de  $n/4$  monotonies de longueur 4
  3. Continuer jusqu'à obtenir une seule monotonie de longueur  $n$

Pourquoi n'est il pas conseillé de réaliser le tri fusion sur fichier avec seulement 2 fichiers?

# Tri fusion sur fichier

- Approche bottom-up avec 3 fichiers qui seront des supports extérieurs distincts
- **Phase d'éclatement**: on répartit les monotonies contenues dans le fichier X dans deux fichiers A et B
  - 1 sur 2 va dans le fichier A
  - 1 sur 2 va dans le fichier B
- **Phase de fusion**:
  - on fusionne la 1<sup>ère</sup> monotonie du fichier A avec la 1<sup>ère</sup> monotonie du fichier B, en écrivant la monotonie résultante dans le fichier X (on écrase l'ancien contenu du fichier X)
  - de même avec les 2<sup>ème</sup> monotonies des fichiers A et B
  - et ainsi de suite jusqu'à la fin des fichiers A et B
- On recommence les phases d'éclatement et de fusion jusqu'à ce que l'on obtienne une seule monotonie



# Tri fusion: exemple

