6. Problèmes avancés de l'algorithmique et de la programmation

Algorithme type de traitement de données : tri à bulles
Complexité des algorithmes
Qualité des programmes
Optimalisation du code
Paradigmes et langages de programmation

Algorithmes type : le tri

- Notion du tri en génie informatique
  - **Trier** : « Classer, répartir les différents éléments d'un ensemble selon quelque critère » ; **Tri** : « Action, manière de trier, de classer : **Le tri de fiches en ordre alphabétique**. »
  - ordre croissant [4 3 5 2 1] → [1 2 3 4 5]
  - ordre descendant [4 3 5 2 1] → [5 4 3 2 1]
- Le **tri à bulles** ou **tri par propagation** est un des algorithmes du tri, qui consiste à faire remonter progressivement les plus petits éléments d'une liste vers son début, comme les bulles d'air (étant plus légères) remontent à la surface d'un liquide
- L'idée de cet algorithme (tri dans l'ordre croissant)
  - On parcourt la liste et on compare des couples d'éléments successifs
  - Lorsque deux éléments successifs ne sont pas rangés dans l'ordre croissant, ils sont intervertis (échangés)
  - Après chaque parcours complet de la liste, on le reprend de nouveau
  - Lorsque aucune interversion n'a lieu pendant un parcours, cela signifie que la liste est triée, l'algorithme est fini

Tri à bulles – analyse

- Début (entrée) : 5 1 4 2 8
- Résultat désiré : 1 2 4 5 8
- Premier parcours :
  - (5 1 4 2 8) → (1 5 4 2 8)

    Ici, on compare les deux premiers éléments (5 et 1) et on les intervertit
  - (1 5 4 2 8) → (1 4 5 2 8)
  - (1 4 5 2 8) → (1 4 2 5 8)
  - (1 4 2 5 8) → (1 4 2 5 8)

    Maintenant que ces deux éléments (5 et 8) sont triés (rangés), on ne les intervertira plus

Tri à bulles – analyse (suite)

- Deuxième parcours :
  - (1 4 2 5 8) → (1 4 2 5 8) On recommence les mêmes actions
  - (1 4 2 5 8) → (1 2 4 5 8)
  - (1 2 4 5 8) → (1 2 4 5 8)
  - (1 2 4 5 8) → (1 2 4 5 8)
  - Nous pouvons constater que la liste est triée en jetant un coup d'œil
  - L'ordinateur est incapable de le faire d'un coup (de même l'humain si la liste est très longue...) ; il doit réexaminer la liste couple par couple
  - Cela peut être achevé avec un parcours normal de l'analyse ; s'il y a 0 interversions, alors chaque élément est bien rangé par rapport au précédent, ce qui veut dire que la liste entière est triée
- Troisième parcours :
  - (1 2 4 5 8) → (1 2 4 5 8) On recommence les mêmes actions
  - (1 2 4 5 8) → (1 2 4 5 8)
  - (1 2 4 5 8) → (1 2 4 5 8)
  - (1 2 4 5 8) → (1 2 4 5 8)
  - Aucune interversion n'a été nécessaire alors on constate que la liste est triée et la procédure peut être terminée

Formalisation de l'algorithme

- Données
  - Entrée : v – vecteur de nombres à trier
    Sortie : v – le même vecteur trié
- Pseudo-code

```
faire
    échanges ← 0
    pour n ← 1 à taille(v)-1
        si v(n+1) < v(n)
            temp ← v(n+1)
            v(n+1) ← v(n)
            v(n) ← temp
            échanges ← échanges+1
        fin si
    fin pour
jusqu'à ce que échanges = 0
```

- Réalisation de l'interversion
  - Début du premier parcours de la boucle « pour » : 5 1 4 2 8
  - n ← 1
    si v(2) < v(1) [VRAI]
    temp ← v(2) = 1
    v(2) ← v(1) = 5
    v(1) ← temp = 1
    …
  - Résultat : 1 5 4 2 8
- Une solution incorrecte
  - v(n+1) ← v(n)
    v(n) ← v(n+1)
  - parce que les instructions sont exécutées de façon séquentielle
    v(2) ← v(1) = 5
    v(1) ← v(2) = 5
  - La variable additionnelle temp est donc inévitable

Complexité algorithmique

- Temporelle dans le pire cas : borne supérieure du nombre des actions nécessaires pour traiter un ensemble de n éléments
- Temporelle en moyenne : nombre des actions effectuées en moyenne pour traiter (p. ex. trier) un ensemble de n éléments
  - Donne une bonne idée du temps de l'exécution
  - Permet de comparer de différents algorithmes destinés au même problème
  - Si les ensembles ont une forme particulière (non représentative par rapport à la majorité des $n!$ combinaisons possibles), alors les performances types pourront être très inférieures ou très supérieures
- Spatiale (en moyenne ou dans le pire cas) : représente l'utilisation de la mémoire que nécessitera l'exécution de l'algorithme
  - Dépend souvent du nombre des éléments à traiter
- La complexité T (temps) ou M (mémoire) est exprimée en fonction du nombre des éléments n de l'ensemble de données traité
  - Notation de Landau : « $T(n) = O(n^2)$ » veut dire que la borne supérieure du nombre d'actions dépend du n en puissance maximale de 2

Cas pratique

- Supposons un processeur :
  - fréquence de l'horloge 1 GHz
  - 1 opération primitive par cycle de l'horloge
- Temps d'exécution (t) :

$T(n)$	$n = 10$	$n = 100$	$n = 1000$
n	0,01 µs	0,1 µs	1 µs
$2n$	0,02 µs	0,2 µs	2 µs
$4n$	0,04 µs	0,4 µs	4 µs
$n \log n$	0,01 µs	0,2 µs	3 µs
$n \log^2 n$	0,01 µs	0,4 µs	9 µs
n^2	0,1 µs	10 µs	1 ms
n^4	10 µs	0,1 s	1,7 min
2^n	1,0 µs	$4{,}0 \cdot 10^{13}$ ans	$1{,}1 \cdot 10^{284}$ ans
4^n	1,1 ms	$5{,}1 \cdot 10^{43}$ ans	$3{,}6 \cdot 10^{603}$ ans

- Taille du problème (n) que l'on peut traiter dans un temps donné :

$T(n)$	$t = 1$ s	$t = 1$ min	$t = 1$ h
n	$1{,}0 \cdot 10^{9}$	$6{,}0 \cdot 10^{10}$	$3{,}6 \cdot 10^{12}$
$2n$	$5{,}0 \cdot 10^{8}$	$3{,}0 \cdot 10^{10}$	$1{,}8 \cdot 10^{12}$
$4n$	$2{,}5 \cdot 10^{8}$	$1{,}5 \cdot 10^{10}$	$9{,}0 \cdot 10^{11}$
$n \log n$	$1{,}2 \cdot 10^{8}$	$6{,}1 \cdot 10^{9}$	$3{,}1 \cdot 10^{11}$
$n \log^2 n$	$1{,}9 \cdot 10^{7}$	$7{,}6 \cdot 10^{8}$	$3{,}3 \cdot 10^{10}$
n^2	31 600	245 000	1 900 000
n^4	178	495	1380
2^n	30	36	42
4^n	15	18	21

Cas pratique (suite)

- Si on utilise un processeur 100 fois plus puissant, comment est-ce que change la taille maximale du problème N ?
  - p. ex. fréquence de l'horloge 10 GHz au lieu de 100 MHz

$T(n)$	N devient
n	$100 \cdot N$
$2n$	$100 \cdot N$
$4n$	$100 \cdot N$
$n \log n$	$79 \cdot N$
$n \log^2 n$	$61 \cdot N$
n^2	$10 \cdot N$
n^4	$3{,}1 \cdot N$
2^n	$N + 6{,}6$
4^n	$N + 3{,}3$

- Algorithmes à complexité de $O(n)$, $O(n \cdot \log^a n)$
  - généralement considérés convenables
- $O(n^a)$ $(a > 1)$
  - acceptables pour des ensembles petits de données
- $O(a^n)$
  - à éviter puisque :
  - le temps de calcul accroît trop vite
  - la taille du problème traitable augmente très peu même pour un temps beaucoup plus long et un processeur beaucoup plus puissant

Complexité des algorithmes développés avant

- Sommation des éléments d'un vecteur à n éléments
  - On considère l'algorithme analysé en cours, il est le basique
  - On compte seules les actions répétées
    - supposant un n large, les autres ont peu d'influence sur le temps de l'exécution
  - C'est donc l'addition
  - Elle est exécutée toujours n fois (indépendamment des valeurs des éléments)
  - Le nombre des actions importantes, c'est donc toujours n (au pire, meilleur et moyen)
  - Complexité temporelle : $T(n) = O(n)$
- Recherche du premier zéro dans un vecteur à n éléments
  - On considère l'algorithme analysé en cours – il y en a d'autres
  - L'action principale, c'est la comparaison
  - Meilleur cas : le zéro sur la première position – 1 comparaison, $T(n) = O(1)$
  - Pire cas : le zéro sur la dernière position ou absent – n comparaisons, $T(n) = O(n)$
  - Au moyen, supposant qu'un zéro est toujours présent avec toute position également probable : $(n+1)/2$ comparaisons, $T(n) = O(n)$

Complexité du tri à bulles

- Opérations principales : comparaison des éléments en couples et leur interversion
- Meilleur cas
  - Lorsque la liste d'entrée est déjà triée
  - On doit effectuer $(n-1)$ comparaisons pour savoir que la liste est triée (avec zéro échanges) parce qu'il y a $(n-1)$ couples d'éléments
  - Nombre des comparaisons : $n-1$
  - Complexité temporelle : $T(n) = O(n)$
- Pire cas
  - Lorsque la liste d'entrée est triée en ordre inverse
  - On doit répéter les passages $(n-1)$ fois parce qu'il faut faire remonter l'élément le plus petit, qui est le dernier, à la première position
  - On peut démontrer que le i-ème passage comprend $(n-i)$ interversions
  - Une interversion prend beaucoup plus de temps qu'une comparaison, alors on néglige les comparaisons pour simplicité
  - Nombre des interversions : $(1 + 2 + \ldots + n-2 + n-1) = (n-1)/2 \cdot n = n^2/2 - 1/2$
  - Complexité temporelle : $T(n) = O(n^2)$

Complexité du tri à bulles (suite)

- Au moyen – analyse probabiliste
  - Il y a $n \cdot (n-1)/2$ couples différents car chaque d'entre les n éléments forme un couple avec chaque d'entre les $n-1$ autres éléments mais (x_1, x_2) et (x_2, x_1) représentent en fait le même couple
  - Si les données sont aléatoires, alors en moyenne la moitié d'entre ces couples devront être intervertis et l'autre moitié – non
  - Nombre des interversions : $n \cdot (n-1)/2/2 = n^2/4 - 1/4$
  - Complexité temporelle : $T(n) = O(n^2)$
- Le nombre des échanges des couples d'éléments est indépendant de la manière dont ces échanges sont organisées (ordre etc.)
- C'est un mauvais algorithme de tri (par rapport aux autres connus)
  - En plus, on peut démontrer que la complexité temporelle $T(n)$ ne peut jamais être meilleure (inférieure) que $O(n \cdot \log n)$ pour tout algorithme fondé sur comparaisons et échanges
  - Algorithmes basés sur d'autres principes sont capables d'atteindre une complexité aussi basse que $O(n)$

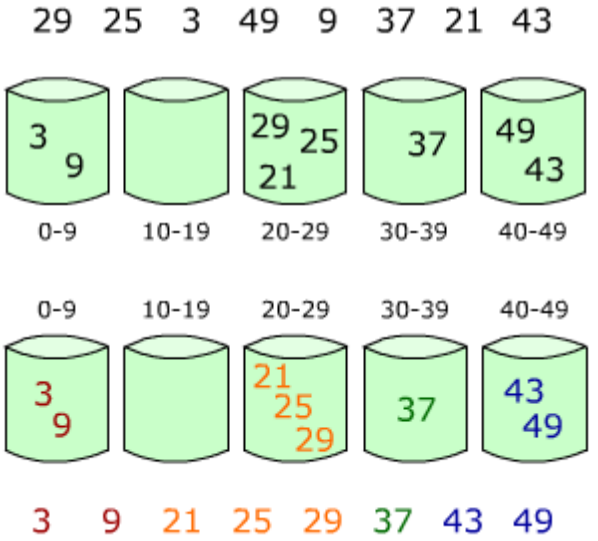
Complexité spatiale

- Afin de la déterminer, on compte :
  - les cellules de mémoire nécessaires pour traiter un ensemble de n éléments
  - en ignorant la donnée d'entrée même (mémoire supplémentaire seulement)
  - seulement les données du même type que la donnée traitée (on ignore les données auxiliaires simples du genre compteur etc.)
- Les algorithmes analysés
  - Sommation
    - 1 telle cellule nécessaire – pour garder la somme
    - alors $M(n) = O(1)$
  - Tri à bulles
    - 1 telle cellule nécessaire – celle qui contient la variable auxiliaire pour effectuer les interversions
    - alors $M(n) = O(1)$
  - Pourtant, en général, beaucoup d'algorithmes de traitement de données possèdent une complexité spatiale de $O(n)$ ou $O(n^2)$

Idées de deux autres algorithmes du tri (à titre d'exemple)

- Tri de Shell
  - Les couples interverties ne doivent pas être des voisins

- Tri par paquets
  - Les éléments sont pré-triés par rangement en plusieurs paquets
  - Ensuite, ils sont triés à l'aide d'un autre algorithme
  - Enfin, ils sont rassemblés
  - Mémoire additionnelle nécessaire pour les paquets



[Source : Pmdumuid, Balu Ertl on Wikimedia Commons]

Comparaison des algorithmes du tri en termes de leur complexité

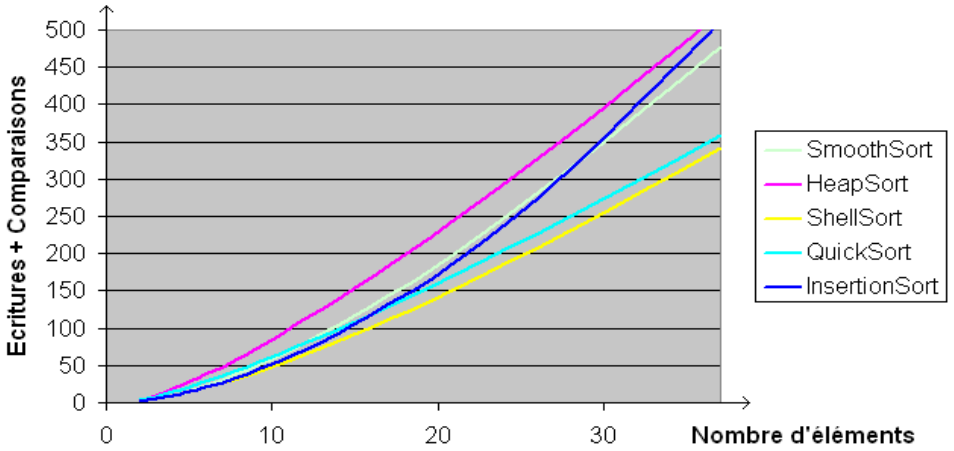
Algorithme du tri	Nom anglais	$T(n)$ au meilleur	$T(n)$ en moyenne	$T(n)$ au pire	$M(n)$	
à bulles	**Bubble**	**$O(n)$**	**$O(n^2)$**	**$O(n^2)$**	**$O(1)$**	
par sélection	Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
par insertion	Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	
de Shell	**Shell**	**$O(n \log n)$**	**$O(n \log^2 n)$**	**$O(n \log^2 n)$**	**$O(1)$**	
rapide (basique)	Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	algorithmes de tri par comparaisons
arborescent (basique)	Binary Tree	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	
par fusion	Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	
Introsort	Introsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	
par tas	Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	
smoothsort	Smoothsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	
par paquets (basique)	**Bucket**		**$O(n)$**	**$O(n^2)$**	**$O(n)$**	algorithmes rapides
par base	Radix		$O(n)$	$O(n)$	$O(n)$	
par dénombrement	Counting		$O(n)$	$O(n)$	$O(n)$	

La complexité des algorithmes rapides dépend aussi de paramètres autres que la taille de la liste, p. ex. du nombre des paquets.

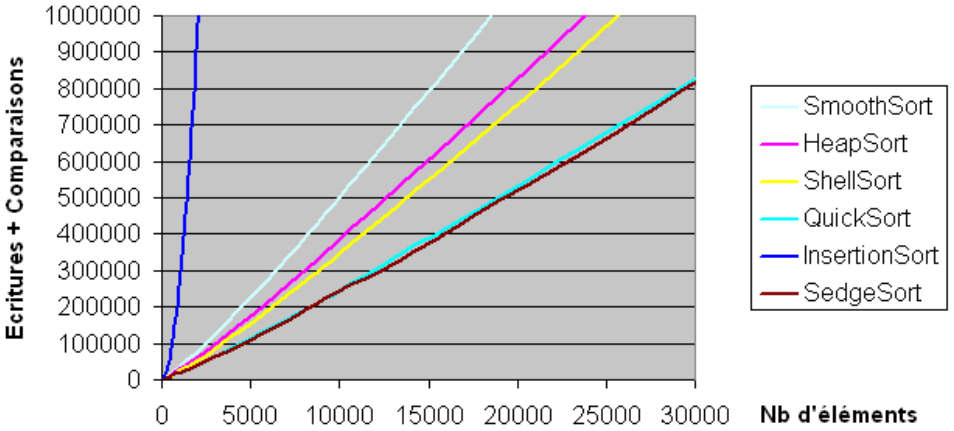
Choix de l'algorithme selon le nombre d'éléments

- Un algorithme peut être favorable pour un nombre d'éléments petit et très défavorable pour un nombre d'éléments élevé
  - Tri par insertion : complexité plus élevée par rapport aux autres pourtant pour des listes courtes, il est autant ou plus efficace
- Parmi les algorithmes d'une même complexité, le nombre d'opération peut être différent
  - La notation de Landau ne prend en compte des coefficients ni des termes d'un degré inférieur au maximal

- Listes courtes



- Listes longues



[Source : Circular sur Wikimedia Commons]

Critères de qualité des programmes (1)

1. Correction (validité)
   - Le programme sert son but : il résout le problème défini en fournissant les résultats corrects pour toutes les entrées prévues dans l'énoncé
   - C'est plutôt l'exigence primaire qu'une qualité
     - Elle est achevée ou pas ; un programme ne peut pas être *plus* ou *moins* valide
   - Elle ne peut pas être supposée
     - C'est naturel que le programmeur a *voulu* créer un programme correct
     - Cependant il n'est pas évident qu'il y ait *réussi*
   - Le programme doit être **testé** : on l'utilise avec de différentes entrées (combinaisons d'arguments = **vecteurs de test**) et on vérifie si les résultats sont corrects
     - L'ensemble des entrées possibles est très souvent illimité
     - Impossible de prouver la correction, seulement de prouver le contraire
     - L'ensemble des vecteurs de test utilisé en tests doit être le plus représentatif possible (ça fait l'objet de la recherche théorique)
     - Si on n'arrive pas à prouver que le programme est incorrect alors il est probable qu'il est correct

Critères de qualité des programmes (2)

2. Sécurité
   - Le programme ne doit jamais permettre de :
     - recevoir ou faire plus que nécessaire vu le rôle de l'utilisateur donné
       - voir des données confidentielles
       - changer des données qu'il est seulement autorisé de voir
     - effectuer des actions n'ayant rien à faire avec son but
       - p. ex. avec des données d'entrée nuisibles fabriquées spécialement à cet effet (pages web, attachements courriel Word avec macros...)
   - On a établi toute une liste de pratiques et concepts de programmation connus pour rendre les programmes vulnérables aux attaques
     - Elles sont à éviter bien que certaines soient très populaires
   - On applique :
     - identification et gestion de rôles (p. ex. système Windows)
     - chiffrement de données – en mémoire ainsi que lors de leur transmission sur les réseaux (p. ex. page web d'une banque)...
   - On a démontré que les autres critères y contribuent aussi

Critères de qualité des programmes (3)

3. Fiabilité
   - Le programme doit réaliser sa tâche correctement
     - C'est équivalent à *correction* mais fiabilité, c'est plus que ça :
   - Il ne doit jamais faire ce qui est indésirable :
     - terminer soudainement
     - planter le processeur ou le système d'exploitation
     - causer le redémarrage soudain de l'ordinateur
     - endommager les données...
   - Tout cela s'applique aussi aux conditions imprévues ou anomales sans faute du programme même (robustesse) :
     - les données ne sont pas correctes, sont trop larges etc.
     - le disque dur est endommagé alors il n'est pas possible d'enregistrer
     - l'utilisateur à tapé une touche par hasard...
   - Il faut alors informer l'utilisateur, suggérer, demander la décision ou confirmation, essayer de remédier le problème automatiquement...
   - Appliquer les mécanismes de gestion d'erreurs
   - On a démontré que les autres critères y contribuent

Critères de qualité des programmes (4)

4. Efficacité économique
   - Basse complexité de l'algorithme en termes du temps et de mémoire
   - Mise en œuvre de l'algorithme possiblement courte en termes de la longueur du code mais surtout du temps d'exécution
     - Minimiser le nombre d'opérations (instructions)
     - Il faut peser les instructions et se concentrer à celles qui prennent le plus du temps à exécuter – la multiplication coûte plus du temps que l'addition, la division plus que la multiplication, opérations matricielles plus que simples, opérandes réelles plus que entières (dans la plupart des langages)
     - Ne pas répéter les mêmes calculs : une fois le résultat obtenu, il vaut mieux le sauvegarder dans une variable et réutiliser ensuite (ça va coûter un peu de la mémoire mais économiser beaucoup de temps)
     - Il faut alors trouver un équilibre (compromis) entre le temps et la mémoire
     - Les bonnes pratiques de programmation et codage, liées aux critères suivants, l'aident

Critères de qualité des programmes (5)

5. Maintenabilité
   - Améliorer le programme, ajouter des fonctionnalités nouvelles etc. ne doit pas poser de problèmes ni au programmeur individuel/original ni à ses collaborateurs/successeurs
   - Aspect important en évaluation de la valeur du travail d'un programmeur
   - Comprend les aspects suivants :

a) Compréhensibilité
   - Documenter, expliquer, mettre des commentaires
   - Attribuer des désignations évocatrices et suivant une convention homogène
   - Éviter trop de niveaux d'imbrication, de structures de données et du code

b) Lisibilité (nécessite a)
   - Éviter des sous-programmes, lignes, expressions, désignations trop longs
     - « Trop », c'est bien relatif ; le programmeur doit réfléchir et décider
   - Utiliser l'indentation correcte (suivre une des conventions reconnues)
   - Découper le code en parties abordables et structurées (programme principal ⟶ fonctions haut niveau ⟶ fonctions élémentaires)
   - Elle peut s'opposer à l'efficacité économique ; un compromis est nécessaire

Critères de qualité des programmes (6)

c) Réutilisabilité (nécessite a+b)

- Il doit être facile d'utiliser le programme ou une partie dans un autre lieu de ce programme ou dans un autre projet car ça permet d'économiser
- Construire le programme de façon modulaire (découper)

d) Extensibilité (nécessite a+b+c)

- Il n'est pas possible de prévoir toutes les fonctionnalités nécessaires et possibles dans le futur
- Souvent il est plus fiable et bénéfique économiquement d'achever d'abord une version limitée, gagner de l'argent et des clients et continuer après
- Il faut prévoir les extensions futures du programme et le créer de façon à faciliter l'introduction d'extensions
- Généraliser, paramétrer plus qu'il paraît momentanément nécessaire

e) Scalabilité

- La performance ne doit pas se détériorer significativement quand la demande augmente (quantité de données, utilisateurs simultanés...)
- Bien choisir l'algorithme
- Utiliser les techniques de programmation avancées appropriées

Critères de qualité des programmes (7)

f) Portabilité

- L'environnement informatique est devenu très diverse
  - systèmes d'exploitation (Windows, MacOS, Linux)
  - mais aussi l'équipement (ordinateurs, smartphones, tablettes)
  - et encore leurs systèmes spécifiques (Android)
- Un logiciel exécutable sur tout système et appareil, ce n'est pas évident
- Langages interprétés : assez simple à condition que des interpréteurs existent pour les différents systèmes
  - cependant les programmes seront longs et lents
  - l'interpréteur occupera en plus de la mémoire et du temps du processeur
- Langages compilés : favorables du point de vue de l'exécution
  - mais il faut compiler le programme pour chaque système séparément (des compilateurs doivent exister)
  - pire encore : le code doit souvent être modifié
  - ça ralentit le développement et le rend plus coûteux (main d'œuvre)
- Réflexion nécessaire au tout début du développement et pourtant déjà avec considération de l'avenir lointain du logiciel (même si au début il est développé pour un seul système et appareil)

Optimalisation d'un programme exemplaire (1)

- Calculer la valeur d'un dépôt à terme de 5 ans, à la fin de chaque année, en considérant une capitalisation mensuelle et une trimestrielle, pour un taux d'intérêt annuel donné
- Donnés d'entrée :
  - depot_init d_0 montant initial du dépôt nombre
  - taux_interet t_a taux d'intérêt annuel en % nombre
- Donnés de sortie :
  - depot_mens d_a valeurs du dépôt, capit. mens. tableau de nombres
  - depot_trim d_a valeurs du dépôt, capit. trim. tableau de nombres
- Formule de calcul :

$$d_a = d_0 \cdot \left[\left(1 + \frac{t_a / 100}{12/m} \right)^{12/m} \right]^a$$

  - où : a – numéro d'année, m – période de capitalisation en mois

Optimalisation d'un programme exemplaire (2)

- Premier essai : long, difficile de saisir l'idée générale

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens(1)=depot_init*(1+(taux_interet/100)/12)^12;
depot_mens(2)=depot_init*(1+(taux_interet/100)/12)^12^2;
depot_mens(3)=depot_init*(1+(taux_interet/100)/12)^12^3;
depot_mens(4)=depot_init*(1+(taux_interet/100)/12)^12^4;
depot_mens(5)=depot_init*(1+(taux_interet/100)/12)^12^5;
depot_trim(1)=depot_init*(1+(taux_interet/100)/4)^4;
depot_trim(2)=depot_init*(1+(taux_interet/100)/4)^4^2;
depot_trim(3)=depot_init*(1+(taux_interet/100)/4)^4^3;
depot_trim(4)=depot_init*(1+(taux_interet/100)/4)^4^4;
depot_trim(5)=depot_init*(1+(taux_interet/100)/4)^4^5;
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

+ correction

Optimalisation d'un programme exemplaire (3)

- Découpage : l'idée générale claire dans *depot.m*

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
calculer_depot_mens;
calculer_depot_trim;
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

► calculer_depot_mens.m

```
depot_mens(1)=depot_init*(1+(taux_interet/100)/12)^12;
depot_mens(2)=depot_init*(1+(taux_interet/100)/12)^12^2;
depot_mens(3)=depot_init*(1+(taux_interet/100)/12)^12^3;
depot_mens(4)=depot_init*(1+(taux_interet/100)/12)^12^4;
depot_mens(5)=depot_init*(1+(taux_interet/100)/12)^12^5;
```

► calculer_depot_trim.m

```
depot_trim(1)=depot_init*(1+(taux_interet/100)/4)^4;
…
depot_trim(5)=depot_init*(1+(taux_interet/100)/4)^4^5;
```

+ compréhensibilité
+ réutilisabilité : on peut copier les procédures de calcul dans un autre programme

Optimalisation d'un programme exemplaire (4)

- Fonctions : éliminent les risques liés à la portée globale des variables désignées avec les mêmes noms dans les 3 fichiers

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot_mens(depot_init,taux_interet);
depot_trim=calculer_depot_trim(depot_init,taux_interet);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

calculer_depot_mens.m

```
function dep=calculer_depot_mens(depot_init,taux_interet)
dep(1)=depot_init*(1+(taux_interet/100)/12)^12;
…
dep(5)=depot_init*(1+(taux_interet/100)/12)^12^5;
endfunction
```

calculer_depot_trim.m

```
function dep=calculer_depot_trim(depot_init,taux_interet)
dep(1)=depot_init*(1+(taux_interet/100)/4)^4;
…
```

\+ fiabilité

Un programmeur qui comprend la portée de variables, peut bien utiliser les mêmes noms de variables dans de différents espaces (c'est l'intérêt même des portées différentes)

Optimalisation d'un programme exemplaire (5)

- Combinaison des deux fonctions en une, la période de capitalisation devenue un troisième argument

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot(depot_init,taux_interet,1);
depot_trim=calculer_depot(depot_init,taux_interet,3);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

► calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital)
dep(1)=depot_init*(1+(taux_interet/100)/(12/periode_capital))...
^(12/periode_capital);
…
dep(5)=depot_init*(1+(taux_interet/100)/(12/periode_capital))...
^(12/periode_capital)^5;
endfunction
```

+ lisibilité : code plus court
+ compréhensibilité : on ne se demande plus pourquoi le codeur a créé 2 fonctions qui font le même
+ réutilisabilité : une fonction universelle
+ extensibilité : autres ou plus de périodes
– lisibilité : une formule répétée encore plus longue

Optimalisation d'un programme exemplaire (6)

- Calculer une fois, sauvegarder dans une variable, l'utiliser plusieurs fois

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot(depot_init,taux_interet,1);
depot_trim=calculer_depot(depot_init,taux_interet,3);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital)
taux_frac=taux_interet/100;
nbre_cap=12/periode_capital;
dep(1)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap;
…
dep(5)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap^5;
endfunction
```

+ lisibilité
− compréhensibilité : on peut facilement oublier ce que représente *nbre_cap*
+ efficacité économique : plus de variables mais moins d'opérations

Optimalisation d'un programme exemplaire (7)

- On décrit la signification des variables à l'aide de commentaires

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot(depot_init,taux_interet,1);
depot_trim=calculer_depot(depot_init,taux_interet,3);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital)
taux_frac=taux_interet/100; # taux d'interet comme fraction
nbre_cap=12/periode_capital; # nombre de capitalisations par an
dep(1)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap;
…
dep(5)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap^5;
endfunction
```

\+ compréhensibilité

Optimalisation d'un programme exemplaire (8)

- Au lieu de répéter les lignes du code, on introduit une structure de répétition

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot(depot_init,taux_interet,1);
depot_trim=calculer_depot(depot_init,taux_interet,3);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital)
taux_frac=taux_interet/100; # taux d'interet comme fraction
nbre_cap=12/periode_capital; # nombre de capitalisations par an
for an=[1:5]
dep(an)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap^an;
endfor
endfunction
```

+ lisibilité : code plus court
+ compréhensibilité : il est devenu évident qu'une même formule est utilisée pour dans chaque année
− lisibilité : il n'est pas évident que dep(an)… est répété 5 fois

Optimalisation d'un programme exemplaire (9)

- Indentation du contenu de la fonction et de la structure de contrôle

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
depot_mens=calculer_depot(depot_init,taux_interet,1);
depot_trim=calculer_depot(depot_init,taux_interet,3);
disp("Avec capitalisation mensuelle : "); disp(depot_mens);
disp("Avec capitalisation trimestrielle : "); disp(depot_trim);
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital)
    taux_frac=taux_interet/100; # taux d'interet comme fraction
    nbre_cap=12/periode_capital; # nombre de capitalisations par an
    for an=[1:5]
        dep(an)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap^an;
    endfor
endfunction
```

+ lisibilité
+ compréhensibilité

Optimalisation d'un programme exemplaire (10)

- Et si, dans le futur, on voudrait changer le nombre d'années considérées ? ou ajouter plus de périodes de capitalisation ?

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
for periode_capital=[1,3]
    dep=calculer_depot(depot_init,taux_interet,periode_capital,5);
    printf("Avec capitalisation tous les %d mois : %.2f\n", periode_capital, dep);
endfor
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,...
periode_capital,nbre_annees)
    taux_frac=taux_interet/100; # taux d'interet comme fraction
    nbre_cap=12/periode_capital; # nombre de capitalisations par an
    for an=[1:nbre_annees]
        dep(an)=depot_init*(1+taux_frac/nbre_cap)^nbre_cap^an;
    endfor
endfunction
```

+ extensibilité : nombre d'années facilement modifiée (c'est par ailleurs grâce à l'introduction de la boucle *for*)
+ extensibilité : un seul nombre à ajouter pour chaque nouvelle période de capitalisation

Optimalisation d'un programme exemplaire (11)

- Multiplication prend moins du temps qu'élévation à une puissance

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
for periode_capital=[1,3]
    dep=calculer_depot(depot_init,taux_interet,periode_capital,5);
    printf("Avec capitalisation tous les %d mois : %.2f\n", periode_capital, dep);
endfor
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital,nbre_annees)
    taux_frac=taux_interet/100; # taux d'interet comme fraction
    nbre_cap=12/periode_capital; # nombre de capitalisations par an
    for an=[1:nbre_annees]
        dep(an)=depot_init*(1+taux_frac/nbre_cap)^(nbre_cap*an);
    endfor
endfunction
```

\+ efficacité économique : temps de calcul plus court

Optimalisation d'un programme exemplaire (12)

- Encore moins d'opérations arithmétiques qui sont répétées pour chaque année : une multiplication seulement

+ efficacité économique : temps de calcul plus court
+ correction : pour nbre_annees=1 ça marche car pour [2:1] la boucle n'est pas exécutée (il a fallu s'en assurer)

depot.m

```
depot_init = input("Montant initial : ");
taux_interet = input("Taux d'interet : ");
for periode_capital=[1,3]
    dep=calculer_depot(depot_init,taux_interet,periode_capital,5);
    printf("Avec capitalisation tous les %d mois : %.2f\n", periode_capita
endfor
```

calculer_depot.m

```
function dep=calculer_depot(depot_init,taux_interet,periode_capital,nbre_annees)
    taux_frac=taux_interet/100; # taux d'interet comme fraction
    nbre_cap=12/periode_capital; # nombre de capitalisations par an
    mult=(1+taux_frac/nbre_cap)^nbre_cap; # multiplicateur chaque annee
    dep(1)=depot_init*mult;
    for an=[2:nbre_annees]
        dep(an)=dep(an-1)*mult;
    endfor
endfunction
```

Optimalisation d'un programme exemplaire (13)

- Comparaison du temps de calcul
  - On peut l'évaluer seulement sur un grand nombre d'opérations
  - Offres pour 1000 clients de la banque, 20 dépôts initiaux, durée de 40 ans
  - On élimine l'affichage pour qu'il n'affecte pas le temps (c'est assez lent)
  - Les fonctions d'Octave tic et toc permettent de mesurer le temps de calcul : *toc* renvoie le temps qui est passé depuis *tic*

depot_test.m

```
tic();
for client=[1:1000]
      for depot_init=[200:200:4000]
            for periode_capital=[1,3,6,12]
                  dep=calculer_depot(depot_init,taux_interet,periode_capital,40);
            endfor
      endfor
endfor
toc()
```

version finale : 40,5 secondes
dep(an)=dep(an-1)*mult

version de départ : 46,7 secondes soit +15 %
dep(an)=depot_init*(1+(taux_interet/100)/(12/periode_capital))...
^(12/periode_capital)^an