Introduction à l'Algorithmique II

Chapitre

Complexité et Preuve d'Algorithme

A.U: 2018 - 2019

Exemple: calcul de 2^n

```
// ************ Solution 1 ******************
fonction puiss_rec1(n:entier): entier
si (n = 0)
retourner 1
sinon
retourner (puiss_rec1(n-1) + puiss_rec1(n-1))
Finsi
Fin
// ************ Solution 2 ******************
fonction puiss_rec2(n:entier):entier
Si (n = 0)
retourner 1
sinon
retourner 2*puiss_rec2(n-1)
Finsi
Fin
```

$2^{n+1} - 2$ APPELS À LA RÉCURSIVITÉ

n APPELS À LA RÉCURSIVITÉ

Complexité

- Q: Quelle est la meilleure solution ?
- Q: meilleure par rapport à quoi ?

- La solution d'un problème d'algo n'est pas unique
  → Plusieurs propositions, **mais** sans doute, une est plus pratique ou **meilleure** !!

- Comparer différents algorithmes (résolvant le même problème)
  - **rapidité :** combien de temps ?
  - **taille des ressources :** combien d'espace mémoire ?
  - peut-on comparer **objectivement** des algorithmes ?
- **Q**: Comment évaluer un algorithme ?

  R: Mesurer sa **complexité**

Mesure de Complexité Algorithmique

- Le temps d'exécution d'un programme dépend de la taille des données.
- On note T(n) le temps d'exécution ou la complexité algorithmique d'un programme portant sur des données de taille n.
- Dans la suite, la complexité d'un algorithme désigne le **nombre d'opérations élémentaires** (affectations, comparaisons, opérations arithmétiques) effectuées par l'algorithme.
- Elle s'exprime en fonction de la taille n des données.

Mesure de Complexité Algorithmique

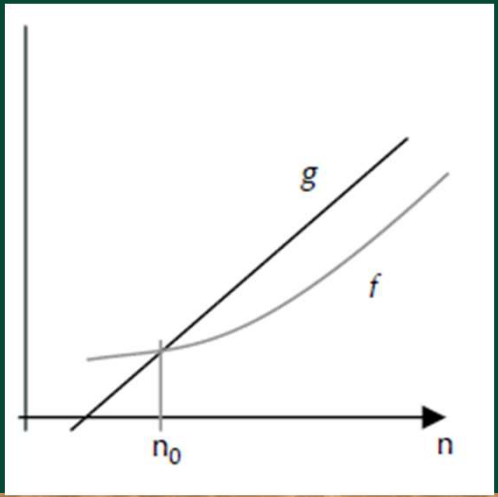
- Deux types de mesures de complexité

1. Pire des cas (*worst case):*
   - *" Quand on s'attend au pire, on n'est jamais déçu"*
   - *O*n définit T(n) comme la complexité maximum, sur tous les ensembles de données possible de taille n.
2. Complexité moyenne (*average),* $T_{moy}(n)$.

- Le complexité "pire des cas" est la plus employée. (Adoptée dans ce cours)

Complexité: notation de landau 'O'

- La notation de Landau "O" : (~Edmund Landau 1877 -1938: wikipedia)
- `Grand Omicron (big Omicron): dit grand O`
- `Comparaison asymptotique: pour des valeurs très grandes (vers l'infini)`
- Pour deux fonctions *f et g on dit que la fonction f est* ***un grand O*** *de la* fonction *g* ***ssi*** *f est dominée asymptotiquement par g.*
- *On note que* $f = O(g)$ *ou* `f(n) = O(g(n))` s'il existe une constante positive c et un entier positif n_0 tel que $f(n) \leq c*g(n)$ pour tout $n \geq n_0$



Complexité: notation de landau 'O'

- Exemples: $f(n)=(n+1)^2$ et $f(n)= 3n^3+2n^2$

1. Soit la fonction $f(n)=(n+1)^2$ pour $n\geq 0$, alors la fonction $f(n)$ est un $O(n^2)$ pour $n_0=1$ et $c=4$.

   En effet, pour $n\geq 1$, on a $(n+1)^2\leq 4n^2$

2. La fonction $f(n)= 3n^3+2n^2$ est $O(n^3)$ avec $n_0=1$ et $c=5$.
   En effet, $3n^3+2n^2-5n^3=2n^2(1-n)\leq 0$ pour $n\geq 1$;
   par conséquent $f(n)$ est $O(n^3)$.

Complexité: Evaluation d'un Algorithme

- n_1 actions élémentaires de genre 1 (par exemple affectations)
  n_2 actions élémentaires de genre 2 (par exemple additions)
  ….
  n_k actions élémentaires de genre k
- Chaque n_i demandant un temps t_i , le temps total d'exécution:

$$t = \sum_i t_i n_i$$

- Pour une machine donnée, si c_2 = max t_i alors $T(n) \leq c_2 \sum_i n_i$

- Donc $T(n) = O\left(\sum n_i\right)$ indépendamment de la machine utilisée.

Complexité: classe

Notation	Type de complexité
$O(1)$	**complexité constante (indépendante de la taille de la donnée)**
$O(\log(n))$	**complexité logarithmique**
$O(n)$	**complexité linéaire**
$O(n\log(n))$	**complexité quasi-linéaire**
$O(n^2)$	**complexité quadratique**
$O(n^3)$	**complexité cubique**
$O(n^p)$	**complexité polynomiale**
$O(n^{\log(n)})$	**complexité quasi-polynomiale**
$O(2^n)$	**complexité exponentielle**
$O(n!)$	**complexité factorielle**

$\log n \ll n^{1/2} \ll n \ll n(\log n) \ll n^2 \ll n^3 \ll 2^n \ll n!$

Complexité: Exemples

- La somme des entiers allant de 1 à n

```
variables n, i, somme : entiers
début
écrire("Donner n :")                    /* 1 écriture */
lire(n)                                 /* 1 lecture */
somme  ← 0                              /* 1 affectation */
pour i allant de 1 à n                  /* 1 affectation, n+1 comparaisons,
   somme ← somme + i                     n additions, n affectations */
FinPour                                 /* n additions, n affectations */
écrire("la somme est :",somme)          /* 1 écriture */
Fin
```

- Total **5n+6 instructions élémentaires**

Complexité:

- Supposons que :
  - l'affectation, la lecture, l'écriture et l'addition prennent chacune un temps t_1
  - la comparaison prend un temps t_2.

- Le temps nécessaire pour la réalisation de cet algorithme est :
  - $(5+4n)t_1 + nt_2 = n(4t_1+t_2)+ 5t_1$.
  - D'où la complexité T(n) est en O(n).

Complexité: recherche ds tableau

```
Fonction recherche(n: entier, Tab[]:entier, x:entier):entier
variables i : entier
début
   i ← 0
   Tant que (i < n et Tab[i] != x) faire
        i ← i+1
   FinTantque
   si(i<n) alors retourne(i)
   sinon retourne(-1)
fin
```

Complexité: recherche ds tableau

- Recherche dichotomique
- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche séquentielle en utilisant la méthode de recherche dichotomique

- **Principe :** diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur x à chaque étape de la recherche. Pour cela on compare x avec T[milieu] :

  – Si x < T[milieu], il suffit de chercher x dans la 1ère moitié du tableau entre (T[0] et T[milieu-1])

  – Si x > T[milieu], il suffit de chercher x dans la 2ème moitié du tableau entre (T[milieu+1] et T[N-1])

Complexité: recherche dichotomique ds tableau

```
inf ← 0 , sup ← N-1, Trouvé ← Faux
TantQue ((inf <=sup)  ET (Trouvé=Faux))
 milieu←(inf+sup)/2
    Si (x=T[milieu]) alors
       Trouvé ← Vrai
    Sinon Si (x>T[milieu]) alors
             inf ← milieu + 1
          Sinon
             sup ← milieu - 1
          FinSi
    FinSi
FinTantQue
Si Trouvé alors
   écrire ("x appartient au tableau")
Sinon
   écrire ("x n'appartient pas au tableau")
FinSi
```

Complexité: recherche dichotomique ds tableau

Exemple de récursivité: recherche dichotomique dans un **tableau trié**: retourne la position de elem s'il existe et -1 sinon

```
fonction rech_dich_rec(T[]:tableau entier, elem:entier,
   deb:entier, fin : entier): vide
variable milieu : entier
Début
si fin < debut
retourne -1
milieu ← div(deb+fin,2)
si elem = T[milieu] retourne milieu
   sinon si elem <  T[milieu]
         retourne rech_dich_rec(T,elem,deb,milieu-1)
         sinon retourne rech_dich_rec(T,elem,milieu +1,fin)
         finsi
Finsi
Fin
```

Complexité: recherche dichotomique ds tableau

- Exemple n = 8
- Pire des cas 3 recherches

- Exemple n = 16
- Pire des cas 4 recherches

- Exemple n = 32
- Pire des cas 5 recherches

- D'où:

  $T(n) = O(\log n)$

Rappel: Tableau 1D-Tri par selection

- À partir du 1 er élement du tableau
- Rechercher le plus petit élement dans le sous tableau à droite de T[i+1 … n-1]
- Si i <> pos_min permuter (T[i] , T[pos_min])

Rappel: Tableau 1D-Tri par selection

- Initial

- Itération 1

- Itération 2

- Itération 3

- Itération 4

Rappel: Tableau 1D-Tri par selection

```
Variable T[N],i,j,k,c,pos_min : Entier
Début
Lire T
Pour i allant de 0 à N-2 faire
    pos_min ← i
    Pour j allant de i+1 à N-1 faire
       Si T[j]<T[pos_min] alors
       pos_min ← j
       Finsi
    FinPour
    Si pos_min <> i alors
       c ← T[pos_min];
       T[pos_min] ← T[i];
       T[i] ← c;
    Finsi
FinPour
Fin
```

Rappel: Tableau 1D-Tri par insertion

- Méthode incrémentielle
- À partir du 2ème élement du tableau (s'il existe !)
- Pour chaque élement T[i] :

  (insérer T[i] dans le sous tableau T[0 … i-1] déjà trié)

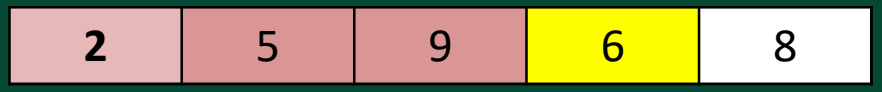
  - ✓ s ← T[i];
  - ✓ Supposer le sous tableau à gauche de T[i] déjà trié
  - ✓ Chercher la position j où T[j-1] ≤ s et T[j] > s
  - ✓ Faire un décalage (d'une case à droite) des élements du sous tableau à droite de j
  - ✓ Insérer l'élement s à la position j

Rappel: Tableau 1D-Tri par insertion

- Initial

- Itération 1

- Itération 2



- Itération 3

- Itération 4

Rappel: Tableau 1D-Tri par insertion

```
Variables T[N],i,j,x : Entier
Début
Lire T
Pour i allant de 1 à N-1 faire
   x ← T[i]          /* insértion de T[i] */
   j ← i
   Tantque j>0 AND T[j-1]>x Faire
       T[j] ← T[j-1]
       j ← j-1
   FinTanque
   T[j] ← x
FinPour
Fin
```

Décalage vers la droite jusqu'à la position d'insertion

Question: l'utilisation de la variable x est elle nécessaire ?

Rappel: Tableau 1D-Tri par bulles

- Pour chaque itération :
  - Parcourir le tableau et comparer les couples d'éléments successifs.
  - Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont permutés.
  - Lorsqu'aucun échange n'a eu lieu pendant un parcours, arrêter (le tableau est trié).

Rappel: Tableau 1D-Tri par bulles

- itération 1:

5	**9**	2	7	6
5	**2**	**9**	7	6
5	2	**7**	**9**	6
5	2	7	**6**	**9**

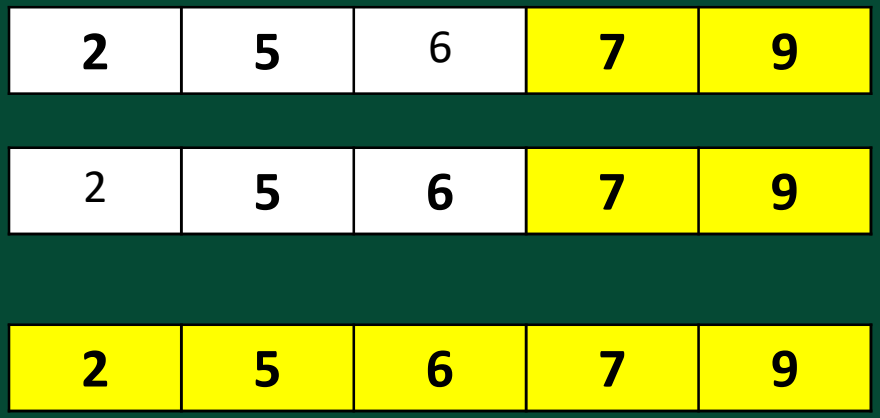
Rappel: Tableau 1D-Tri par bulles

- itération 2:

5	**2**	7	6	**9**
2	**5**	**7**	**6**	**9**
2	5	**6**	**7**	**9**

Rappel: Tableau 1D-Tri par bulles

- Itération 3:



- Tri par bulle, appelé aussi tri par propagation

Rappel: Tableau 1D-Tri par bulles

```
variable Tableau T[N], i,j,k,c,echange : Entier
Début
k ← N-1
echange ← 1
Répéter
echange ← 0
Pour i allant de 0 à k-1 faire
    si T[i] > T[i+1] alors
      c ← T[i]
      T[i] ← T[i+1]
      T[i+1] ← c
      echange ← 1
    Finsi
FinPour
K ← k-1
jusqu'à echange = 0
```

Complexité: Tris des tableaux

- Exemples:
  - Tri par selection:
  - $O(n^2)$

  - Tri par insertion
  - $O(n^2)$

  - Tri par bulle
  - $O(n^2)$

Chapitre : Preuve d'algorithme

Preuve d'Algorithme: Correction Partielle

- **Définition.1** Un algorithme est **partiellement correct ssi**, lorsqu'il s'arrête, il a fait ce qu'il doit faire.
- Exemple:

```
Algorithme: racine carré aléatoirement
Variable a,n,RacineCarree : Entier
Début
    lire(a)
    n ← 0
    tant que ( n*n <> a )
    n ← random(0,a) /* nombre entier aléatoire entre 0 et a */
    fin tant que
    RacineCarree ← n
Fin
```

Preuve d'Algorithme: Correction Partielle

- Cet algorithme est partiellement correct:
- il ne s'arrête que si la condition n*n <> a est fausse,
  càd si n*n = a est vérifiée ;
  le résultat RacineCarree = n est donc égal à la racine carée de a.
- **il se peut** qu'un algorithme partiellement correct **ne soit pas satisfaisant**, car on n'a pas la garantie qu'il s'arrêtera.
- Dans l'exemple précédent, il ne s'arrêtera jamais, si le nombre a n'est pas le carrée d'un entier.

Preuve d'Algorithme: Correction

- **Définition.2**: Un algorithme est **correct** ssi, il est partiellement correct et il s'arrête nécessairement, lorsque les données initiales vérifient sa pré-condition (conditions que doivent remplir les entrées valides de l'algorithme) ;
- on parle de **terminaison**.
- Dans ce cas, on est sûr qu'il fera ce qu'il doit faire.

- On n'a imposé aucune contrainte sur le nombre d'itérations possibles, ni sur la place de mémoire disponible pour stocker des variables.

- Lors de l'implémentation, cela pourra durer arbitrairement longtemps, ou devenir impossible à cause du manque de la mémoire.

Preuve d'Algorithme: Correction

- Exemple d'un algorithme correct: chercher **la partie entière de la racine carrée** d'un réel a positif.

```
Algorithme
Variable n,RacineCarree : Entier
         a              : Réel
Début
   n ← 0
   TantQue ( n*n ≤ a )
      n ← n + 1
      Fin TantQue
      RacineCarree ← n-1
Fin
```

Preuve d'Algorithme

- Comment montrer qu'un algorithme est correct ?
- si :
  - il s'arrête,
  - pour toute entrée, il produit le résultat attendu.

- On peut :
  - tester quelques entrées → prouver qu'il est incorrect (mais pas le contraire)
  - Il est mieux de prouver logiquement que l'algorithme est correct :
    - terminaison : variant de boucle,
    - résultat attendu : invariant de boucle.

Preuve d'Algorithme: Variant de boucle

- On appelle variant de boucle, une expression qui:
  - est un entier positif tout au long de la boucle,
  - décroît strictement.

- Lorsqu'une suite d'entiers $\{u_i\}$ décroît strictement, il existe un rang N à partir duquel les termes u_i sont négatifs.

- Dans la boucle, $u_i > 0$ → l'algorithme termine nécessairement

- Le variant de boucle est souvent le simple contenu d'une variable telle qu'un compteur de boucle.

Preuve d'Algorithme: Variant de boucle

- Exemple:

Algorithme 1 : Calcul de k^n

Entrées : entier n, réel k

Sorties : réel puissance

```
c=n
puissance=1
tant que c>0 faire
    puissance=puissance*k
    c=c-1
retourner puissance
```

- Montrons que l'algorithme s'arrete:

Dans la boucle, c est définie par la suite $\{c_i\}$ telle que dans la boucle:

$c_0 = n$

$c_{i+1} = c_i - 1 \Rightarrow \forall i, c_{i+1} < c_i$

$\forall i, c_i > 0$

La suite $\{c_i\}$ est entière, positive et strictement décroissante, donc l'algorithme s'arrete.

Démonstration par récurrence

- Démonstration par récurrence de la propriété P_n
  - Initialisation: Montrer que Pn est vraie à partir d'un certain rang n_0
  - Hérédité: Montrer que $P_n \Rightarrow P_{n+1}$

- Exemple:

Montrer par récurrence que la suite définie par :

$$\begin{cases} u_0 = 2 \\ u_{n+1} = \frac{1}{3} u_n \end{cases}$$

s'écrit $u_n = \frac{2}{3^n}$.

- La preuve par récurrence est aussi appelée **preuve par induction**

Invariant de Boucle

- Pour démontrer que l'algorithme produit l'effet attendu, on utilise un invariant de boucle, càd une propriété ou une expression qui :
  - est vérifiée avant d'entrer dans la boucle,
  - reste vraie après chaque itération de la boucle,
  - dont la valeur au rang n (à la sortie de la boucle), est la propriété qu'on veut démontrer.
- Démarche similaire au raisonnement par récurrence.
- **Difficulté** : trouver une expression susceptible d'être un invariant de boucle.

Invariant de Boucle

Un exemple :

Algorithme 2 : Calcul de k^n

Entrées : entier n, réel k

Sorties : réel p

```
c=n
p=1
tant que c>0 faire
    p=p*k
    c=c-1
retourner p
```

Dans la boucke, deux suites $\{p_i\}$ et $\{c_i\}$ sont définies par récurrence:

$p_0 = 1$ et $p_{i+1} = k * p_i$

$c_0 = n$ et $c_{i+1} = c_i - 1$

On veut montrer qu'en sortie de la boucle $p = k^n$.

Montrons la proposition Pr_i: $p_i = k^{n-ci}$ est un invariant de boucle.

Si Pr_i est un invariant de boucle alors à la sortie de la boucle

$c = c_n = 0$ et $p = p_n = k^n$ (cqfd)

Invariant de Boucle

Montrons la proposition $\mathcal{P}_i : p_i = k^{n-c_i}$.

Initialisation : pour $i = 0$, $p_0 = 1 = k^0$ et $c_0 = n$ donc $p_0 = k^{n-c_0}$. $\mathcal{P}_0$ est vérifiée.

Récurrence : $\mathcal{P}_i : p_i = k^{n-c_i}$ supposé vraie. Montrons alors que $\mathcal{P}_{i+1} : p_i = k^{n-c_{i+1}}$ est vraie.

$p_{i+1} = k * p_i = k * k^{n-c_i} = k^{n-c_i+1}$

Or $c_{i+1} = c_i - 1$ donc :

$p_{i+1} = k^{n-c_{i+1}}$: $\mathcal{P}_{i+1}$ est vraie.

A la sortie de la boucle $\mathcal{P}_n$ est vraie donc $p_n = k^{n-c_n}$.

Or $c_n = 0$ donc $\boxed{p = p_n = k^{n-c_n} = k^n}$.CQFD !!!

Application

```
Données : un entier naturel a et un entier naturel n
Résultat : un nombre p
variables b,m,p : Entiers
p ← 1
b ← a
m ← n
Tant que m > 0 Faire
si m est impair alors
p ← p*b
FinSi
b ← b*b
m ← div(m,2)
Fin Tant que
```

1. Donner la valeur finale de la variable p lorsque (a ; n) = (3; 6) et (a ; n) = (4; 5).
2. Justifier que l'algorithme se termine. Quelle est la valeur de m à la fin de la boucle ?
3. Vérifier que «$pb^m = a^n$» est un invariant de boucle.

References

- Introduction à l'algorithmique. Cours et exercices. Cormen et al. 2e édition.
- Algorithmique Raisonner pour concevoir. Christophe HARO.
- Éléments d'algorithmique. D. Beauquier et al.
- *Informatique pour tous. P. Chatel.*
- Algorithmique, M. El Marraki.
- http://pise.info/algo/