ENSAM
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

Algorithmique Avancée

Animé par : Dr. ibrahim GUELZIM
Email : ib.guelzim@gmail.com

References

- Introduction à l'algorithmique. Cours et exercices. Cormen et al. 2e édition. (En 3rd Edition)
- Algorithms, FOURTH EDITION, Robert Sedgewick and Kevin Wayne. Princeton University.
- Algorithmique, M. El Marraki.
- https://jeffe.cs.illinois.edu/teaching/algorithms/

Sommaire

- Rappels
  - Introduction et notions générales
  - Analyse et conception d'algorithmes
  - Complexité d'algorithmes classiques : 3 Tris de tableaux, 2 recherches dans un tableau, Schéma de Hörner
  - Preuves d'algorithmes
- Autres algorithmes de tri :
  - Tri par fusion
  - Tri par Tas
- Complexité moyenne :
  - Application au Tri rapide
  - Algorithmes sur des structures de recherches spéciales :
    - Table de Hachage et Fonction de Hachage,
    - Bloom Filter,
    - Count Min Sketch
- Programmation dynamique
- Traitements de chaines de Caractères :
  - Recherche de chaine de caractères
  - Compression de données
- Transformée de Fourier et applications

Envers un algorithme

Algorithme

- Terminaison ?
- Produit résultat attendu ?
- Réalisable ? (complexité)
- Peut on faire mieux ? (optimalité)

Algorithmes

- Algorithmique :
  - Hommage à الخوارزمي [780 – 850] :
    Savant mathématicien perse (Khawarezm ∈ Khorasan → Baghdad pendant la dynastie abbaside)
  - Alkhawarizmi → Algoritmi →Algorithmique
  - **Art** de décrire une tâche (chemin, recette…)
  - **Art** de mettre les idées en ordre: penser avant d'agir.
  - Mise de **l'ordre** dans la pensée: gain considérable de temps.

Algorithmes

- Procédure de calcul

- une **suite d'étapes** dont le but est de **décrire** la résolution d'un problème ou l'accomplissement d'une **tâche**.

- une **méthode de résolution** de problème énoncée sous la forme d'une **série d'opérations** à effectuer dans un **ordre** bien défini.

- Procédé permettant de résoudre un problème en un **nombre fini** d'opérations.

- une suite finie et **non ambigüe** d'opérations permettant de résoudre un problème.

Algorithmes

- Procédure de calcul

- une **suite d'étapes** dont le but est de **décrire** la résolution d'un problème ou l'accomplissement d'une **tâche**.

- une **méthode de résolution** de problème énoncée sous la forme d'une **série d'opérations** à effectuer dans un **ordre** bien défini.

- Procédé permettant de résoudre un problème en un **nombre fini** d'opérations.

- une suite finie et **non ambigüe** d'opérations permettant de résoudre un problème.

Algorithmes

- Q: Utilité des algorithmes ?!
- R: exemples
  - Internet: Recherche de routes optimales pour l'acheminement des données.
  - Commerce électronique: La cryptographie s'appuient sur des algorithmes numériques pour préserver la confidentialité. Background :
    - Signature Numérique
    - Fonction de Hashage
    - Théorie des nombres
  - Une compagnie pétrolière veut savoir où placer ses puits de façon à maximiser les profits,
  - Compression de données,
  - …

Analyse et conception

- Analyser un algorithme :
  - Prévoir les ressources nécessaires à cet algorithme.
    - la mémoire,
    - le processeur,
    - mais, souvent, c'est le temps de calcul qui nous intéresse.
- Plusieurs algorithmes peuvent en résulter

  → éliminer les algorithmes inférieurs et garder la meilleure solution.
- Analyse du meilleur cas, du plus défavorable, ou du cas moyen ?

Conception

- **Méthode incrémentale**: utilise un processus **itératif** où chaque itération augmente la quantité d'information.
- Ex: Tri par insertion
  - insérer T[i] dans le sous tableau T[0 ... i-1] (déjà trié)
  - Exemple : insérer 8 dans [2 6 9 15]

2	6	9	15	8	1	41	7
2	6	9	15	15	1	41	7
2	6	9	9	15	1	41	7
2	6	8	9	15	1	41	7
2	6	8	9	15	1	41	7

8

Décaler les éléments opportuns à droite

2	6	9	15	8	1	41	7
2	6	9	8	15	1	41	7
2	6	8	9	15	1	41	7
2	6	8	9	15	1	41	7

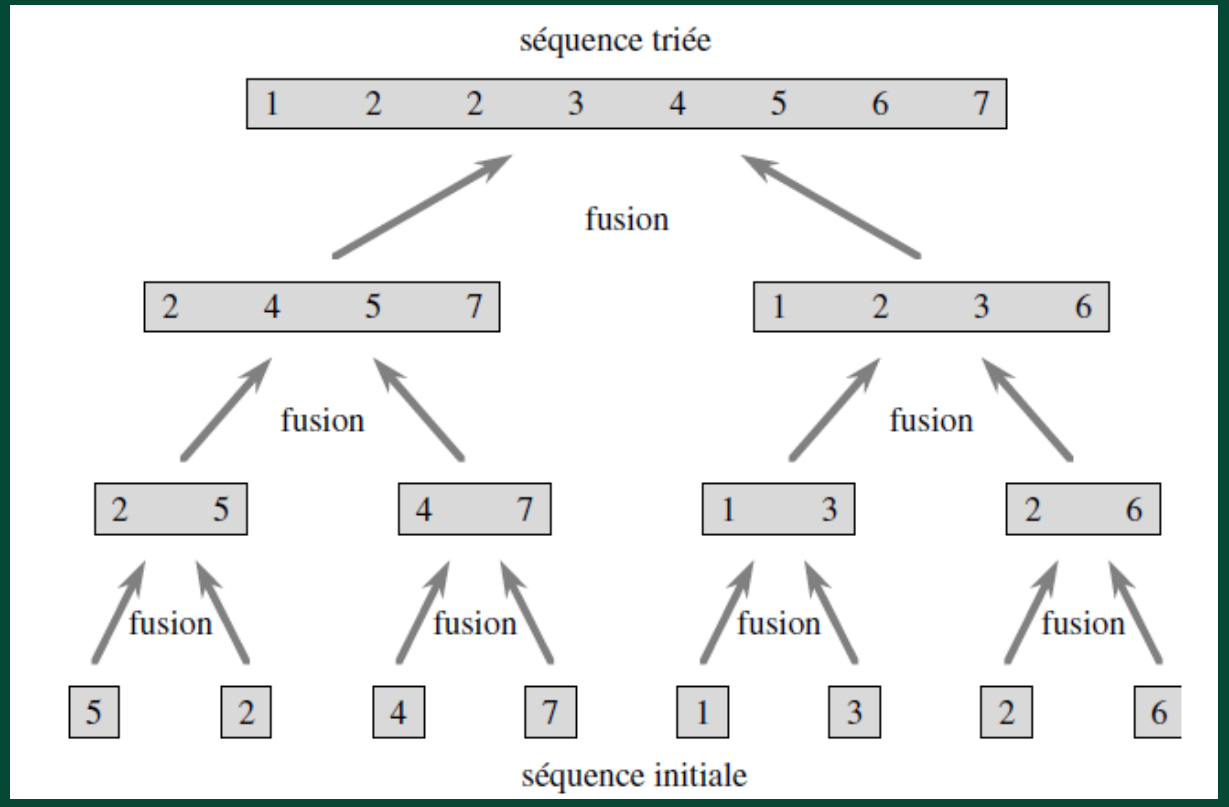
Permuter les éléments opportuns

Conception

- **Diviser pour régner**:
  - Analyse Descendante :
    - Décomposer chaque Problème "Composé" en sous Pb, jusqu'à l'arrivé à des sous Pb élémentaires (non décomposables). (appelés : feuilles de l'arbre)
    - Si phase finale (indissociable - indivisible) on passe à la réalisation.
  - Analyse Ascendante:
    - Si phase NON FINALE : Reconstitution de la solution de chaque Pb en regroupant les solutions de ses sous Pb.
    - Le résultat est fournis à la phase d'avant : père (niveau plus haut dans l'arborescence)

Conception

- **Diviser pour régner**:
  - Ex: Tri par fusion (cf. plus loin)



Fonctions Récursives

- Introduction: Il arrive, en mathématique, que des suites soient définies de la manière suivante :

  u_0 = constante

  u_n = f(u_{n-1})

- Exemple :
  - La suite factorielle : n! = n*(n-1)!, pour n≥1 avec 0!=1, peut s'ecrire :

    ```
    f(0) = 1
    f(n) = n*f(n-1)
    ```

  - Ce que l'on peut traduire par : f(n) = (si n=0 alors 1 sinon n*f(n-1)).

Fonctions Récursives

- Cela peut se traduire en algorithmique par :

```
fonction factorielle_rec(n :entier) : entier
début
si (n=0) alors retourne(1)
sinon retourne(n*factorielle_rec(n-1))
finsi
Fin
```

- Dans la fonction factorielle_rec(), on constate que la fonction s'appelle elle-même.
- Ceci est possible, puisque la fonction factorielle_rec() est déclarée avant son utilisation (c'est l'entête d'une fonction qui la déclare).

Fonctions Récursives

- Que se passe-t-il lorsque on calcul factorielle_rec(3) ?

```
Factorielle_rec(3) = 3 * factorielle_rec(2)
                   = 3 * (2*factorielle_rec(1))
                   = 3 * (2 * (1*factorielle_rec(0)))
                   = 3 * (2 * (1 * (1)))
                   = 3 * (2 * (1))
                   = 3 * (2)
                   = 6
```

Fonctions Récursives

- La récursivité est un concept fondamental en mathématiques et en informatique.

- Un programme récursif est un programme qui s'appelle lui-même.

- Pour éviter la **boucle infinie**, il faut que le programme cesse de s'appeler lui-même.

  → Un programme récursif doit contenir une condition **d'arrêt** (ou de terminaison) qui autorise le programme à ne plus faire appel à lui-même.

- Fonction récursive:
  - Directe: Contient appel à elle même
  - Indirecte (croisée): contient un appel à une fonction qui emmène à l'appel de la fonction initiale.

Fonctions Récursives

- **Souvent** la récursivité dans les fonctions est ainsi:

```
fonction Fonc_Rec(paramètres) : Type_retoure
début
   Si (cond arret 1 vérifiée)
     retourner valeur_1
   FinSi
   appel(s) récursivité Fonc_Rec(paramètres')
Fin
```

- Attention, une mauvaise condition d'arrêt → Solution fausse.

- Une fonction récursive peut avoir **plusieurs conditions d'arrêt** .
  Exemple: Fonction de Fibonacci

Fonctions Récursives

- Suite de Fibonacci

$F_n = F_{n-1} + F_{n-2}$, pour n ≥2 avec F_0=0 et F_1=1

- On a un programme récursif simple associé a cette relation :

```
fonction Fibo1(n entier) : entier
début
  si (n = 0 ou n = 1) alors retourne(n)
  sinon retourne(Fibo1(n-1) + Fibo1(n-2))
  finsi
Fin
```

- Le nombre d'appels nécessaires au calcul de F_n est égal au nombre d'appels nécessaires au calcul de F_{n-1} plus celui relatif au calcul de F_{n-2},

- ceci correspond bien à la définition de la suite de Fibonacci .

Fonctions Récursives

- La récursivité
  - Est un concept proche de l'esprit humain, **MAIS**
  - n'est pas nécessairement la **meilleure** solution de résolution d'un Pb en terme de temps d'execution ou d'espace mémoire réservé (**complexité**)
- Exemple: 2ème algorithme de calcul de **Fn** (en utilisant un tableau)

```
constante : N = 26
fonction Fibo2(entier :n) : entier
variable i: entier ; tableau F[N] : entier
Début
  F[0]  ← 0
  F[1]  ← 1
  pour i allant de 2 à n faire
       F[i]  ← F[i-1] + F[i-2]
  Finpour
  retourne F[n]
Fin
```

Fonctions Récursives

Exemple de récursivité: Recherche Dichotomique Récursive RDR dans un **tableau trié**: retourne la position de elem s'il existe et -1 sinon

```
fonction RDR(T[]:entier,elem:entier,Deb:entier,Fin : entier):Entier
variable milieu : entier
Début
si Fin < Deb
retourne -1
milieu ← (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
```

Exemple de 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"*
   - On définit T(n) comme la complexité maximum, sur tous les ensembles de données possible de taille n.

2. Complexité moyenne (*average*), $T_{moy}(n)$.

3. Meilleure des cas

- Le complexité "pire des cas" est la plus employée.

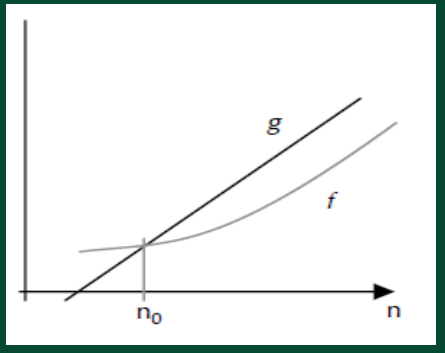
Complexité: notation de landau 'O'

- 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 $n \geq n_0$

- On dit *que* $f = \Omega(g)$ *ou* $f(n) = \Omega(g(n))$ si $g = O(f)$
- On dit *que* $f = \Theta(g)$ *si* $f = O(g)$ *et* $g = O(f)$



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 $T(n) = O(\sum_i n_i)$ indépendamment de la machine utilisée.

Complexité: classes

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, p entier positif
$O(n^{\log(n)})$	complexité quasi-polynomiale
$O(c^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

- $\log^*(n) << \log n << n^{1/2} << n << n.\log(n) << n^2 << n^3 << 2^n << n!$
- $\log^*(n)$ est le logarithme itéré: 0 si $n \leq 1$ et $1 + \log^*(\log(n))$ sinon
- nombre d'itération que le log doit être appliqué avant que le résultat soit inférieur ou égal à 1

Complexité: Exemple

- 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,
                                            n additions, n affectations */
  somme ← somme + i                     /* n additions, n affectations */
FinPour
écrire("la somme est :",somme)          /* 1 écriture */
Fin
```

- Total 5n+6 instructions élémentaires

Complexité: Exemple

- 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 séquentielle

- 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 trié ↗

```
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 trié ↗

```
// fonction 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
FinSi
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 trié ↗

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

Complexité: Schéma de Hörner

- Problème :
- On considère le polynôme en x réel, a coefficients réels, de degré n : $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

- On veut calculer sa valeur $P(x_0)$ pour $x = x_0$ donné, en utilisant les seules opérations élémentaires : addition et multiplication.

Complexité: Schéma de Hörner

- 1ère méthode : On peut écrire un algorithme qui calcule $a_n x_0^n$, $a_{n-1} x_0^{n-1}$,…, $a_1 x_0 + a_0$ les unes après les autres et les additionne.

- Calculons en fonction de n le nombre d'opérations élémentaires qui seront effectuées lors de l'execution de cet algorithme :
  - nombre de multiplications pour un $a_i x^i$: i
  - nombre de multiplications pour tous :
    $1 + 2 + \ldots + (n-1) + n = n(n+1)/2$
  - nombre d'additions : n
  - Total : $n(n+3)/2$.

- Donc cet algorithme est $O(n^2)$.

Complexité: Schéma de Hörner

- 2ème méthode: P(X) peut s'écrire:

$P(X) = (...((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + ... + a_1)x + a_0$

→ On peut écrire $P(x_0)$ de la manière suivante:

Analyse: itérer n fois:

- Multiplier A par x_0 et additioner le coefficient suivant
- Mettre le résultat obtenu dans A

Commencer avec $A = a_n$

Réalisation

```
Entrée : a[n+1], x0 : Entier
variable A,i : Entier
A ← a[n]
Pour i allant de n-1 à 0 pas:-1 faire
  A ← A * x0 + a[i]
FinPour
```

Complexité: Schéma de Hörner

- 2ème méthode: P(X) peut s'écrire:

$$P(X) = (...((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + ... + a_1)x + a_0$$

Complexité:

- n multiplications
- n additions
- n+3 lectures
- n affectations
- 1 écriture

- Donc il est O(n)
- Meilleur que le précédent (pour de grandes valeurs de n).

Rappel: Tableau 1D-Tri par selection

- À partir du 1er 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
```

Tri par insertion : Conception

- Ex: Tri par insertion
  - insérer **T[i]** dans le sous tableau

    T[0 ... i-1] (déjà trié)

15	9	2	6	8	1	41	7
15	9	2	6	8	1	41	7
9	15	2	6	8	1	41	7
2	9	15	6	8	1	41	7
2	6	9	15	8	1	41	7
2	6	8	9	15	1	41	7
1	2	6	8	9	15	41	7
1	2	6	8	9	15	41	7
1	2	6	7	8	9	15	41

Conception

- Ex: 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

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).

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**

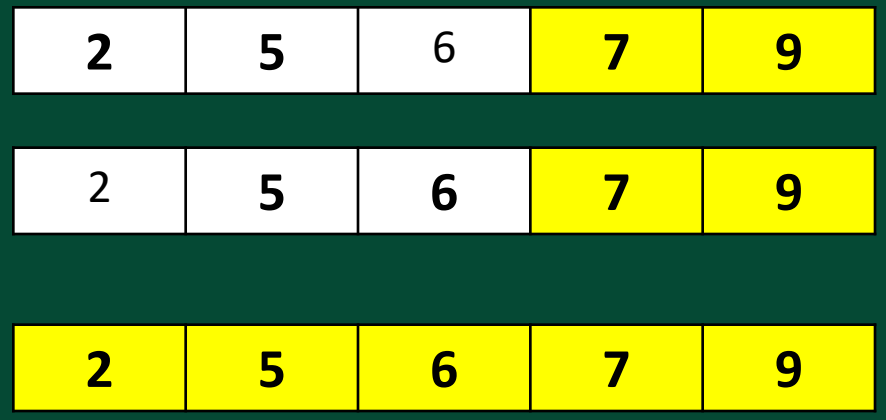
Tri par bulles

- itération 2:

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

Tri par bulles

- Itération 3:



- Tri par bulle, appelé aussi tri par propagation

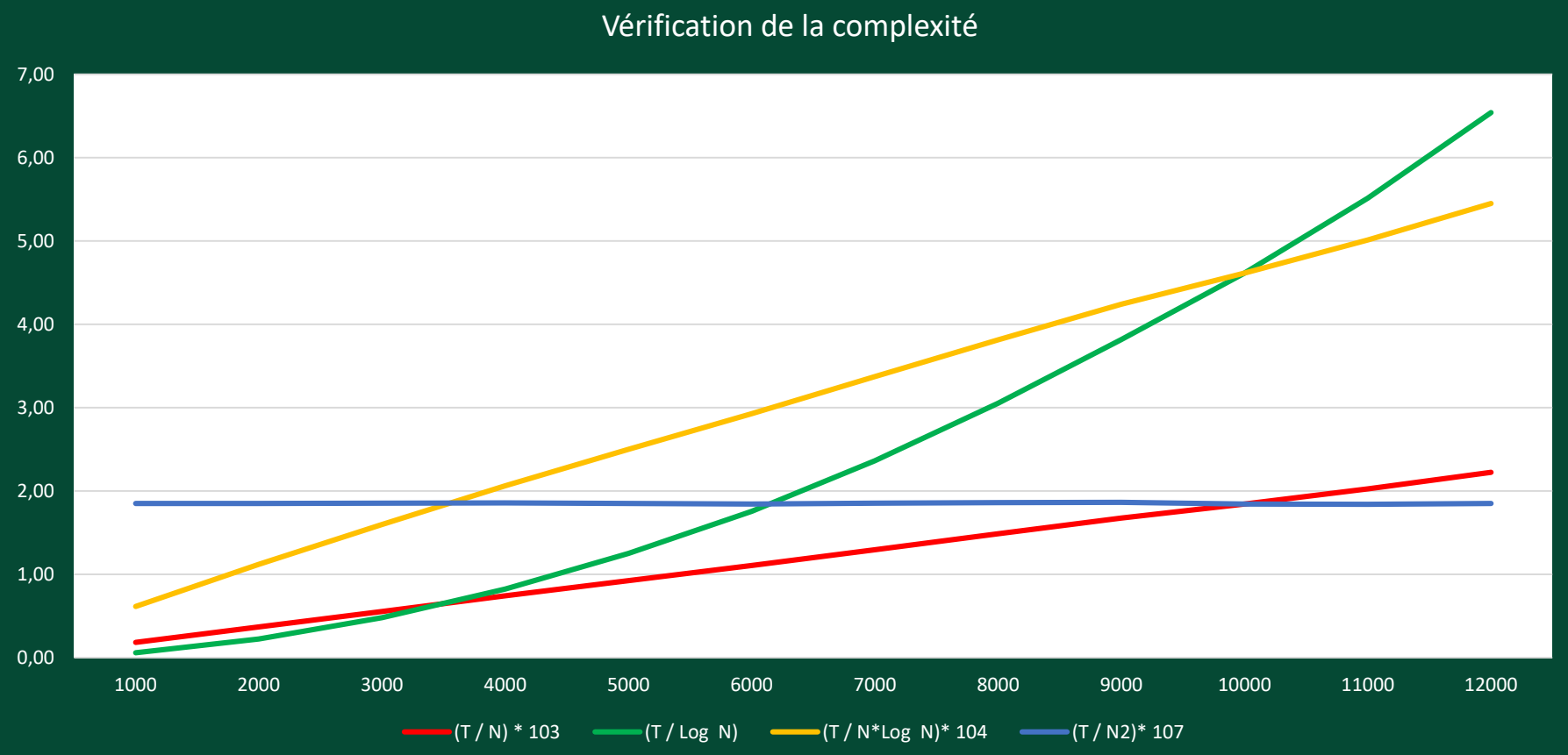
Complexité: Tris des tableaux

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

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

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

Vérification de la complexité



Partie : 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 : Calcul de K^n
Entrées  n : Entier
         k : Réel
Variable c : Entier
Début
c ← n
p ← 1
TantQue c > 0
     p ← p*k
     c ← c - 1
FinTantQue
Fin
```

- 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 P_n 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:
  - $U_0 = 2$
  - $U_{n+1} = (1/3)U_n$

  s'écrit $U_n = 2/3^n$

- 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: Exemple

```
Algorithme : Calcul de Kⁿ
Entrées  n : Entier
         k : Réel
Variable c : Entier
Début
c ← n
p ← 1
TantQue c > 0
     p ← p*k
     c ← c - 1
FinTantQue
Fin
```

Dans la boucle, 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

Dans la boucle, 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.
  - Vrai pour i = 0
  - Mq par recurrence que $p_{i+1} = k^{n-Ci+1}$
    - $p_{i+1} = k * p_i$ (par construction)
    - or $p_i = k^{n-Ci}$ (par supposition) , d'où
    - $p_{i+1} = k * k^{n-Ci} = k^{n - Ci - 1}$ (or par construction $C_i - 1 = C_{i+1}$)
    - $p_{i+1} = k^{n-Ci+1}$
- Puisque Pr_i est un invariant de boucle, alors vrai à la sortie de la boucle
- Or $C_n = 0$ et $p = p_n = k^{n - Cn} = k^n$ (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.

FIN