Dr. Mahfoud Houari
mahfoud.houari@gmail.com
Université Abou-Bakr Belkaïd - Tlemcen
2022/2023

Position du cours :

- C'est la suite du cours *Algorithmique et structures de données*
- **Ce qui est préalablement acquis :**

1. La ***récursivité*** (un premier pas vers la conception algorithmique).

2. Définir et manipuler des ***structures de données*** (séquentielles et hiérarchiques).

3. Proposer des programmes « ***coûte que coûte*** » pour des problèmes classiques (tri, recherche, fusion,…).

4. Quelques notions de complexité.

Position du cours :

Votre bagage est ***nécessaire*** mais pas ***suffisant*** :

1. Il y a d'autres méthodes de conception autre que la ***récursivité***.
2. La ***récursivité*** elle-même existe en *deux modes*.
3. Le choix de la structure de données influence sur la qualité de votre programme.
4. Les problèmes classiques n'offrent pas de job.
5. En pratique, c'est plutôt l'***efficacité*** de la solution qui importe ***et non pas sa correction***.
6. Étude avancée de la complexité algorithmique.

Objectif principal du cours :

Apprendre à proposer un/le « Bon Algorithme » pour un problème donné

Concevoir un *bon* algorithme –
via *un bon choix de la structure de données*

Exemple :

Dans un magasin donné, le manager utilise un tableau ***T*** pour stocker le gain journalier du magasin pendant une période de ***N*** jours. Une petite opération sur ce tableau consiste à calculer le gain total entre deux journées ***d*** et ***f***: c-a-d. La somme des éléments se trouvant dans l'intervalle [***d***, ***f***].

Concevoir un *bon* algorithme – via *un bon choix de la structure de données*

Solution 1

T
T[0]
…
T[***d***]
…
T[***f***]
…
T[***N***-1]

- Parcourir tous les éléments entre ***d*** et ***f*** et faire leur somme.
- *Coût total de la somme:* ***f*** – ***d*** sommes exactement ou encore ***N*** - 1 sommes au pire.
- *Coût total (sommes et accès):* ***f*** – ***d*** + 1 accès, et ***f*** – ***d*** sommes.
- Peut-on faire mieux ?

Concevoir un *bon* algorithme – via *un bon choix de la structure de données*

Solution 2

S
T[0]
T[0]+***T***[1]
T[0]+***T***[1]+***T***[2]
…
T[0]+…+***T***[***d***]
…
T[0]+…+***T***[***f***]
…
T[0]+…+***T***[***N - 1***]

- Utiliser un autre tableau ***S***.
- Aucun parcourt n'est nécessaire.
- *Coût total:* ***S***[***f***] – ***S***[***d***] donnera la somme voulue. Donc 03 opérations au total quelque soit la taille du ***T***.
- *Coût de préparation:* ***N*** cases mémoire de plus.

Concevoir un *bon* algorithme – via *un bon choix de la structure de données*

Synthèse

- Plus que le nombre d'opérations diminue, plus que l'efficacité augmente.
- Le coût de préparation est fait une seule fois donc ça ne compte pas trop.
- Si ***T*** ne change pas fréquemment alors ***Solution 2*** est la meilleure.
- On peut se procurer plus d'espace mémoire pour réduire le temps d'exécution.

Concevoir un *bon* algorithme –
via *une bonne formalisation du problème*

Exemple de problème: *« Formalisation 1 »*

Soient donnés deux tableaux ***T1*** et ***T2*** de taille ***N***, le but est de tester s'il y a une permutation des éléments de ***T1*** qui rendra ***T1*** équivalent à ***T2***.

Solution :

- Considérer une permutation possible ***Permutte***(***T1***);
- Tester si ***Permutte***(***T1***) == ***T2***;
- Si oui, alors afficher ***true***, sinon, répéter le traitement.

⇒ Il y a au maximum ***N!*** permutations possibles.

Concevoir un *bon* algorithme –
via *une bonne formalisation du problème*

Exemple de problème: *« Formalisation 2 »*

Soient donnés deux tableaux ***T1*** et ***T2*** de taille ***N***, le but est de tester si tous les éléments de ***T1*** appartiennent à ***T2***.

Solution :

- Parcourir tous les éléments de ***T1*** et tester si chacun appartient à ***T2***;
- Si oui, alors afficher ***true***, sinon afficher ***false***.

⇒ Il y a au maximum N^2 comparaisons à faire.

Concevoir un *bon* algorithme –
via *une bonne formalisation du problème*

Synthèse :

- *Formalisation 1* nécessite un traitement factoriel
- *Formalisation 2* nécessite un traitement quadratique
- Alors que *Formalisation 1* est équivalente à *Formalisation 2*

Concevoir un *bon* algorithme –
via *un bon paradigme de conception*

Exemple de problème:

Soit donné un tableau ***T*** composé d'entiers triés en ordre croissant, tester s'il y a un indice i tel que ***T*[*i*] == *i***.

Solution brute-force:

```
boolean TestNaif (int[] T){
        for(int i =0 ; i < T.length ; i++){
                if(T[i] == i) return true ;
        }
        return false ;
}
```

N *comparaisons*

Concevoir un *bon* algorithme –
via *un bon paradigme de conception*

Exemple de problème:

Soit donné un tableau ***T*** composé d'entiers triés en ordre croissant, tester s'il y a un indice i tel que ***T*[*i*] == *i***.

Solution brute-force:

```
boolean TestNaif (int[] T){
        for(int i =0 ; i < T.length ; i++){
                if(T[i] == i) return true ;
        }
        return false ;
}
```

N comparaisons

Solution Intelligente :

```
boolean TestOptimise (int[] T){
        int D=0, F= T.length - 1;
        while(D <= F){
                int M = (D + F)/2 ;
                if(T[M] == M) return true ;
                if(T[M] < M) D = M + 1 ;
                else F = M - 1 ;
        }
        return false ;
}
```

$log_2(N)$ *comparaisons*

Objectifs du cours en détails :

1. Arrêter de programmer ***naïvement :***

   ⇒ *Penser à l'efficacité et pas juste à la correction.*

   ⇒ *Suivre les étapes de conception d'algorithmes.*

2. Savoir s'il existe un algorithme pour un problème donné :

   ⇒ *Notion de* ***décidabilité***

   ⇒ ***Exemple :*** *Le «* ***problème de Correspondance de Post*** *» est* ***indécidable****.*

Soient deux listes: L1 et L2

α_1	α_2	α_3
a	*ab*	*bba*

β_1	β_2	β_3
baa	*aa*	*bb*

Une solution existe avec les indices (3, 2, 3, 1) car:

$$\alpha_3\alpha_2\alpha_3\alpha_1 = bba + ab + bba + a = bbaabbbaa = bb + aa + bb + baa = \beta_3\beta_2\beta_3\beta_1.$$

Objectifs du cours en détails :

3. Quel type de solution pour un problème donné ?

⇒ ***Solution correcte :*** la plus ***exacte*** possible
(est-ce possible tout le temps ?)

⇒ ***Solution efficace :*** la plus ***rapide*** possible.
(est-ce possible tout le temps ?)

Exemples :

- *Recherche dans une structure de données (Correcte, Efficace).*
- *Faire des jointures multiples de tables relationnelles (C, ~~E~~).*
- *Établir un emploi du temps (~~C~~, E).*

⇒ ***Bon algorithme » :*** compromis entre *correction* et *efficacité*.
(quelle mesure à sacrifier ?)

Objectifs du cours en détails :

4. Analyser les performances *théoriques* et *pratiques* d'un algorithme.

5. *Classification* des problèmes.

6. Méthodes de conception d'algorithmes :

- *Méthodes exhaustives*
- *Diviser pour régner*
- *Programmation dynamique*
- *Algorithmes Gloutons*

7. Savoir proposer des solutions *approximatives*.

Plan du cours :

1. Analyse de complexité (***noyau du module***)
2. Diviser pour régner
3. Programmation dynamique
4. Algorithmes gloutons
5. Les classes de problèmes ***P***, ***NP*** et ***NPC***
6. Les algorithmes d'approximation

Notion d'efficacité

Exemple 1:

Calculer $\mathbf{X^N}$.

Solution 1 : $\mathbf{X^N = X \cdot X^{N-1} = X \cdot X \cdot X^{N-2} = X \cdot X \cdot X \cdot X^{N-3} \ldots}$

$\Rightarrow$ ***N* - 1** *multiplications possibles*

Notion d'efficacité

Exemple 1:

Calculer X^N.

Solution 2 : $X^N = \begin{cases} X^{N/2} \cdot X^{N/2} & \text{si } N \text{ est pair} \\ X^{N/2} \cdot X^{N/2} \cdot X & \text{si } N \text{ est impair} \end{cases}$

⇒ *2 multiplications aux maximum par niveau*

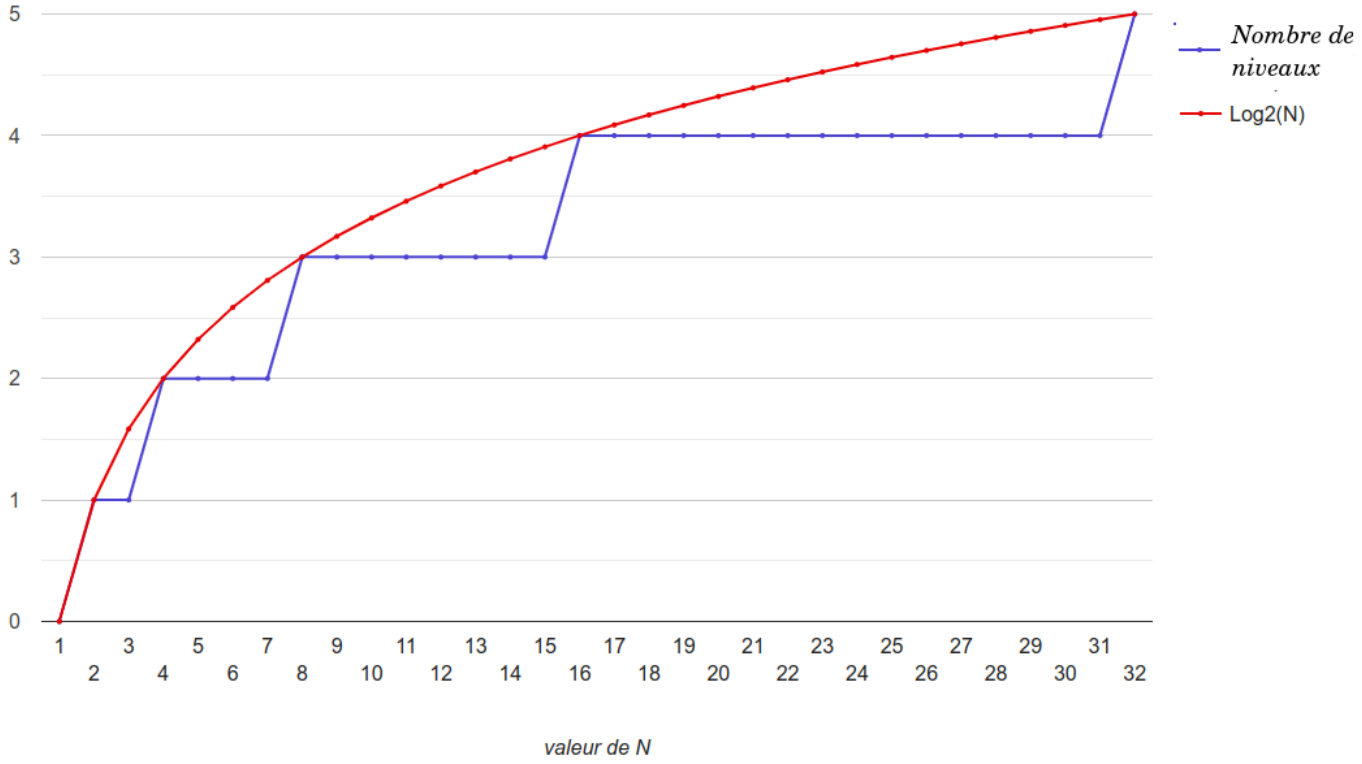
⇒ *combien de niveaux y-a-t il ?*

Notion d'efficacité

Exemple 1:

Calculer X^N.

Solution 2 :



Notion d'efficacité

Exemple 1:

Calculer $\mathbf{X^N}$.

Solution 2 :

⇒ ***2*** *multiplications au maximum par niveau*

⇒ *Nombre de niveaux borné par* ***log2(N)***

⇒ D'où, ***2.log2(N)*** *multiplications au maximum*

Conclusion :

Solution 2 *est plus efficace que* ***Solution 1***

Notion d'efficacité

Exemple 2:

Calculer la somme des éléments d'un tableau.

Solution Itérative :

```
int Iterative (int[] T){
        int S = 0 ;
        for(int e : T)
                S += e ;
        return S;
}
```

N-1 sommes

Notion d'efficacité

Exemple 2:

Calculer la somme des éléments d'un tableau.

Solution Récursive 1 :

```
//Au début : i=0
int RNT (int[] T, int i){
        if(i == T.length -1)
                return T[i] ;
        else
                return T[i] + RNT (T, i+1) ;
}
```

N-1 sommes

Solution Récursive 2 :

```
//Au début : i=0, s=0 ;
int RT (int[] T, int i, int s){
        if(i == T.length)
                return s ;
        else
                return RT (T, i+1, s+T[i]) ;
}
```

N-1 sommes

Notion d'efficacité

Exemple 2:

Calculer la somme des éléments d'un tableau.

Synthèse :

- Les trois méthodes réalisent le même nombre de sommes.
- ***RNT*** nécessite une pile en mémoire pour stocker le résultat de chaque appel récursif. Elle est appelée ***solution récursive non-terminale***.
- ***RT*** nécessite de stocker uniquement le résultat du dernier appel récursif. Elle est appelée ***solution récursive terminale***.
- Une solution récursive non-terminale nécessite plus d'espace mémoire que sa version itérative ou sa version terminale.
- D'où : les méthodes ***Iterative*** et ***RT*** sont plus efficaces, en termes de mémoire, que la méthode ***RNT***.

Efficacité & Optimalité

Définitions :

Un algorithme ***A*** est ***plus efficace*** que ***B*** s'il permet d'exécuter moins d'opérations que ***B***.

Un algorithme est ***optimal*** s'il n'existe pas un autre algorithme plus efficace que lui.

Un algorithme assure le ***passage à l'échelle*** s'il reste ***efficace*** pour n'importe quelle instance.

Notion d'optimalité

Exemple de problèmes

Tester si un élément ***E*** appartient à une matrice symétrique ***M*** de taille ***N.N***.

Solution 1 :

Considérer toutes les cases de ***M***.

⇒ ***N² tests possibles***

Solution non optimale

Solution 2 :

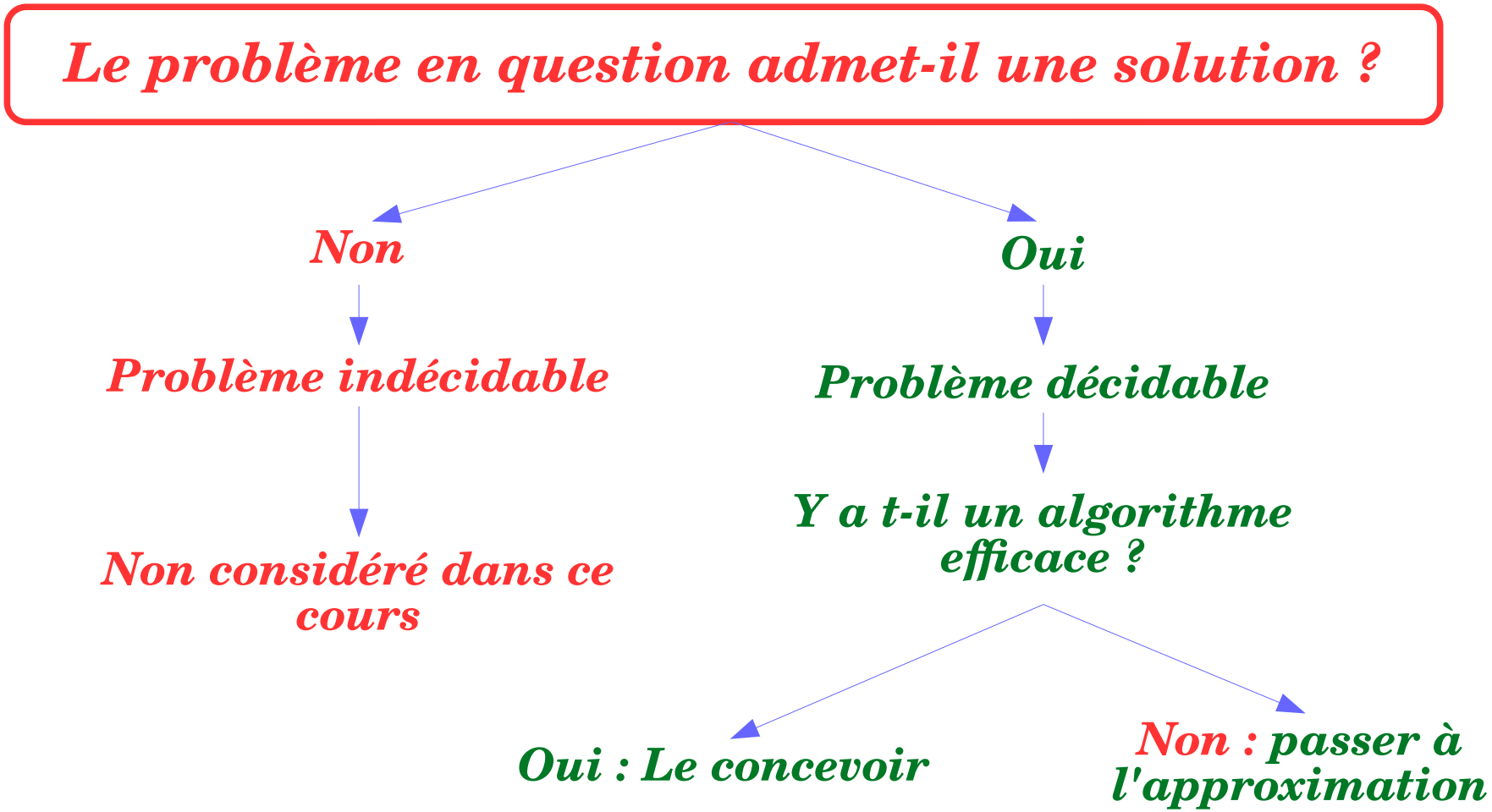
Considérer juste les cases de ***M*** se trouvant au-dessous de la diagonale.

⇒ ***(N² + N)/2 tests possibles***

Solution optimale

Classification des problèmes :

Avant de passer à l'algorithmique, il faut répondre à ça :



Notion de décidabilité :

Définition :

Un problème de décision ***P*** est dit ***décidable*** s'il existe un **algorithme** qui exécute un **nombre fini d'étapes** pour répondre par ***Oui/Non*** à la question posée par ce problème.

Quelques problèmes décidables :

1. Tester si une liste donnée est triée :
   - ***Décidable :*** *Un simple parcours fera l'affaire.*
2. Le plus court chemin entre deux sommets :
   - ***Décidable :*** *Algorithme de Dijkstra par exemple.*
3. Existence d'un cycle dans un graphe :
   - ***Décidable :*** *Adapter l'algorithme BFS pour ça.*
4. Trouver un cycle hamiltonien :
   - ***Décidable MAIS*** *sans algorithme* ***efficace*** *connu (****1 m.$****).*

Quelques problèmes décidables :

5. Le jeu sudoku :
   - ***Décidable MAIS*** *sans algorithme efficace général.*

6. Regrouper les étudiants en trinômes qui s'entendent :
   - ***Décidable MAIS*** *sans algorithme* ***efficace*** *connu (****1 m.$****).*

7. Emploi du temps :
   - ***Décidable MAIS*** *sans algorithme efficace général.*

8. Détection de plagiat :
   - ***Décidable MAIS*** *sans algorithme efficace général.*

Problème de l'arrêt de programmes (*Halting Problem*) :

Définition du problème :

Étant donné un programme ***P*** avec une entrée ***E.*** Tester automatiquement ***si P(E) termine ou boucle infiniment***.

Exemple 1:

```
void Rien (int N){
      while (true){
            N++ ;
            Afficher(N) ;
      }
}
```

Facilement détectable

Exemple 2:

```
boolean Existe (Noeud N, int E){
      if(N.valeur() == E)
            return true ;
      for(Noeud fils : N.fils())
            Existe(fils, E) ;
}
```

La détection dépend de la logique de l'algorithme donc impossible de généraliser

Problème de l'arrêt de programmes (*Halting Problem*) :

Définition du problème :

Étant donné un programme ***P*** avec une entrée ***E.*** Tester automatiquement ***si P(E) termine ou boucle infiniment***.

Résultat négatif :

Alan Turing a prouvé en 1936 qu'il est ***indécidable*** de trouver un **algorithme général** qui permet de **tester**, pour n'importe quel programme ***P*** et n'importe quelle entrée ***E***, si ***P(E)*** s'arrête.