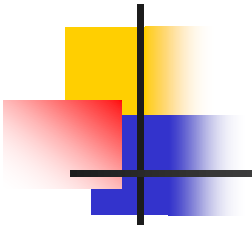


Algorithmique Avancée et Complexité

Cours du Master "Réseaux et Systèmes Distribués" 1ère année

Dr Amar ISLI

Département d'Informatique

Faculté d'Electronique et d'Informatique

Université des Sciences et de la Technologie Houari Boumediène

BP 32, El-Alia, Bab Ezzouar

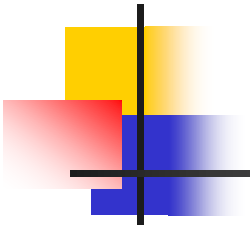
DZ-16111 ALGER

Nouvelle page :

http://www.usthb.dz/perso/info/aisli

Ancienne page :

http://www.usthb.dz/fei-deptinfo/perso/aisli/amarisli.htm



CHAPITRE I
Introduction

Problème et instance d'un problème

- Définition d'un problème

Un problème est une description abstraite à laquelle est associée une question nécessitant une réponse.

CHAPITRE I
Introduction

Problème et instance d'un problème

- Exemple :
  - le problème du voyageur de commerce
  - ou TSP (Traveling Salesman Problem)

  Description : un graphe (non orienté) G=<V,E> dont les arêtes sont étiquetées par des coûts. Les nœuds correspondent aux villes à visiter, et le coût d'une arête peut, par exemple, correspondre à la distance séparant les deux villes qu'elle relie

CHAPITRE I
Introduction

Problème et instance d'un problème

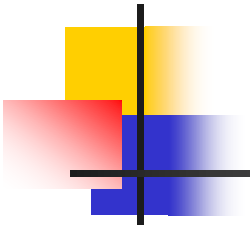
- Exemple :
  - le problème du voyageur de commerce

Question : calcul d'un cycle $(i_1,\ldots,i_n,i_1)$ de coût minimal, partant d'une ville i_1 et visitant chacune des autres villes exactement une fois avant de revenir à la ville de départ i_1.

CHAPITRE I
Introduction

Problème et instance d'un problème

- Instance d'un problème
  - Une instance d'un problème est une spécialisation du problème obtenue en donnant une spécification exacte des données
- Exemple : une instance du TSP
  - G=<V,E> avec
    - V={A,B,C,D}
    - E={(A,B),(A,C),(A,D),(B,C),(B,D),(C,D)}
    - c(A,B)=2; c(A,C)=3; c(A,D)=5; c(B,C)=3; c(B,D)=1; c(C,D)=1



CHAPITRE I
Introduction

Remarque

- Un programme, qui est une implémentation d'un algorithme, est conçu pour un problème donné
- Une exécution d'un programme nécessite comme entrée une instance du problème associé ; en sortie, l'exécution fournit une réponse à la question associée au problème adaptée à l'instance en question

CHAPITRE I
Introduction

Taille d'un problème

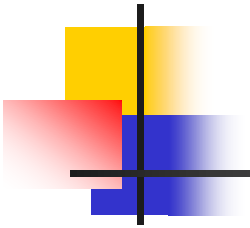
La taille d'un problème est la taille de l'espace mémoire nécessaire à sa représentation ; elle est mesurée en fonction des données du problème

- Exemple : dans le cas du TSP, les données sont
  - Le nombre de villes à visiter
  - Le nombre d'arêtes
  - Les coûts des différentes arêtes

CHAPITRE I
Introduction

Taille d'un problème

- Une instance du TSP peut, par exemple, être représentée par une matrice nxn, n étant le nombre de villes à visiter
- Les éléments d'une telle matrice seront les coûts des différentes arêtes
  - La matrice est symétrique : les éléments (i,j) et (j,i) sont égaux
  - La diagonale de la matrice est à 0 : les éléments (i,i) valent 0
  - Si deux villes i et j ne sont pas reliées par une arête, les éléments (i,j) et (j,i) seront mis à $+\infty$

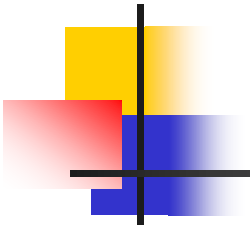


CHAPITRE I
Introduction

Algorithme et complexité

- Algorithme

  Un algorithme est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème



CHAPITRE I
Introduction

Algorithme et complexité

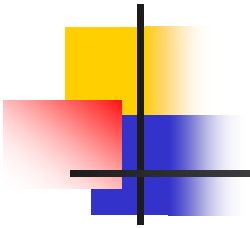
- Complexité
  - La complexité d'un algorithme est la mesure du nombre d'opérations élémentaires qu'il effectue sur le problème pour lequel il a été conçu
  - La complexité est mesurée en fonction de la taille du problème ; la taille étant elle-même mesurée en fonction des données du problème
  - La complexité est par conséquent mesurée en fonction des données du problème

CHAPITRE I
Introduction

Complexité

- Exemple :

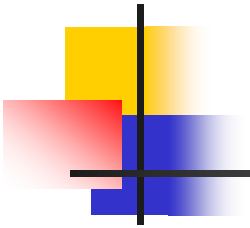
  dans le cas du TSP, la complexité va être mesurée en fonction des données du problème, qui sont
  - Le nombre de villes à visiter
  - Le nombre d'arêtes
  - L'espace nécessaire (en termes de bits par exemple) pour la représentation du coût d'une arête



CHAPITRE I
Introduction

Complexité

- Complexité au meilleur
  - Best-case complexity
- Complexité en moyenne
  - Average-case complexity
- Complexité au pire
  - Worst-case complexity



CHAPITRE I
Introduction

Notations de Landau

- On ne mesure généralement pas la complexité exacte d'un algorithme
- On mesure son ordre de grandeur qui reflète son comportement asymptotique, c'est-a-dire son comportement sur les instances de grande taille

CHAPITRE I
Introduction

Notations de Landau

Besoin des notations asymptotiques de Landau :

- $f=O(g)$ ssi il existe n_0, il existe $c\geq 0$,
  pour tout $n\geq n_0$, $f(n)\leq cxg(n)$
- $f=\Omega(g)$ ssi $g=O(f)$
- $f=o(g)$ ssi pour tout $c\geq 0$, il existe n_0,
  pour tout $n\geq n_0$, $f(n)\leq cxg(n)$
- $f=\Theta(g)$ ssi $f=O(g)$ et $g=O(f)$

CHAPITRE I
Introduction

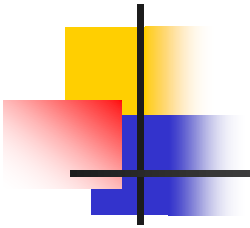
Illustration

- tri par insertion
  - **Entrée :** Un tableau de n nombres
  - **Sortie :** Le tableau trié par ordre croissant

CHAPITRE I
Introduction

Illustration : tri par insertion

```
TRI-INSERTION{
    Pour j=2 à n{
      clé= A[j]
      i=j-1
      Tant que i>0 et A[i]>clé{
                              A[i+1]=A[i]
                              i=i-1
                              }
      A[i+1]=clé
      }
    }
```



CHAPITRE II
Conception d'algorithmes efficaces

A. Piles et files

Définition (pile) :

- Une pile est une structure de données mettant en œuvre le principe « dernier entré premier sorti »
- LIFO : Last In First Out

CHAPITRE II
Conception d'algorithmes efficaces

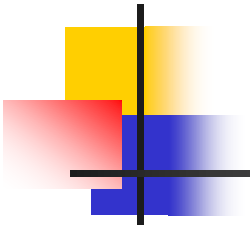
Une pile P peut être implémentée par un tableau, et elle est caractérisée par :

- Un sommet noté sommet(P) indiquant l'indice de l'élément le plus récemment inséré dans la pile
- Un caractère spécial, comme $, initialisant la pile
- Une procédure EMPILER(P,x)
- Une fonction DEPILER(P)
- Une fonction booléenne PILE-VIDE(P) retournant VRAI si et seulement si P est vide

```
PILE-VIDE(P){
    Si sommet(P)=$
       alors retourner VRAI
       sinon retourner FAUX
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

```
EMPILER(P,x){
    Si sommet(P)=longueur(P)
                alors erreur (débordement positif)
                sinon{      sommet(P)=sommet(P)+1
                            P[sommet(P)]=x
                            }
    }
DEPILER(P){
    Si PILE-VIDE(P)
                alors erreur (débordement négatif)
                sinon{      sommet(P)=sommet(P)-1
                            retourner P[sommet(P)+1]
                            }
    }
```



CHAPITRE II
Conception d'algorithmes efficaces

A. Piles et files

Définition (file) :

- Une file est une structure de données mettant en œuvre le principe « premier entré premier sorti »
- FIFO : First In First Out

CHAPITRE II
Conception d'algorithmes efficaces

Une file F peut être implémentée par un tableau, et elle est caractérisée par :

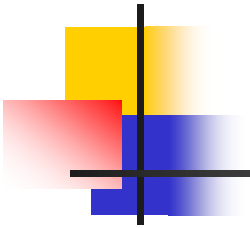
- Un pointeur tête(F) qui pointe vers la tête de la file (l'élément le plus anciennement inséré)
- Un pointeur queue(F) qui pointe vers la première place libre, où se fera la prochaine insertion éventuelle d'un élément
- Initialement : tête(F)=NIL et queue(F)=1

CHAPITRE II
Conception d'algorithmes efficaces

```
FILE-VIDE(F){
    si tête(F)=NIL
      alors retourner VRAI
      sinon retourner FAUX
    }
INSERTION(F,x){
    si [tête(F)≠NIL et queue(F)=tête(F)]
      alors erreur (débordement positif)
      sinon{
                F[queue(F)]=x
                queue(F)=[queue(F)+1](modulo n)
                si[tête(F)=NIL] alors tête(F)=1
                }
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

```
SUPPRESSION(F){
    si FILE-VIDE(F)
      alors erreur (débordement négatif)
      sinon{
                temp=F(tête(F));
                tête(F)=[tête(F)+1](modulo n);
                si[tête(F)=queue(F)]{
                          tête(F)=NIL ;
                          queue(F)=1;
                          }
                retourner temp;
                }
    }
```



CHAPITRE II
Conception d'algorithmes efficaces

A. Piles et files

- EXEMPLES
  - Pile d'exécution
  - Automate à pile : $a^n c b^n$
  - Simulation de la file d'attente d'un magasin

CHAPITRE II
Conception d'algorithmes efficaces

B. Listes chaînées

Définition (liste chaînée) :

- Une liste chaînée est une structure de données dont les éléments sont arrangés linéairement, l'ordre linéaire étant donné par des pointeurs sur les éléments
- Un élément d'une liste chaînée est un enregistrement contenant un champ *clé*, un champ *successeur* consistant en un pointeur sur l'élément suivant dans la liste
- Si le champ *successeur* d'un élément vaut NIL, l'élément est le dernier élément de la liste, appelé également *queue* de la liste
- Un pointeur TETE(L) est associé à une liste chaînée L : il pointe sur le premier élément de la liste
- Si une liste chaînée L est telle que TETE(L)=NIL alors la liste est vide

B. Listes chaînées

- **Exemple**
  - Liste chaînée contenant initialement 20, 7 et 10
    - INSERTION(30)
    - SUPPRESSION(7)

Liste doublement chaînée

Liste chaînée triée

Liste chaînée circulaire (anneau)

CHAPITRE II
Conception d'algorithmes efficaces

B. Listes chaînées

- **Algorithmes de manipulation de listes simplement chaînées**

```
RECHERCHE-LISTE(L,k){
    x=TETE(L);
    tant que(x!=NIL et clé(x)!=k)x=successeur(x);
    retourner x;
    }

INSERTION-LISTE(L,X){
    successeur(x)=TETE(L);
    TETE(L)=x;
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

B. Listes chaînées

- **Algorithmes de manipulation de listes simplement chaînées**

```
SUPPRESSION-LISTE(L,X){
    Si x=TETE(L) alors TETE(L)=successeur(x)
    sinon{
      y=TETE(L);
      tant que successeur(y)<>x faire y=successeur(y);
      successeur(y)=successeur(x);
      }
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

C. Graphes

- **Graphe orienté :** couple (S,A), S ensemble fini (sommets) et A relation binaire sur S (arcs)
  - Boucle (arc reliant un sommet à lui-même)
  - Un arc (u,v) *part* du sommet u et *arrive* au sommet v
  - Degré sortant d'un sommet : nombre d'arcs en partant
  - Degré ®entrant d'un sommet : nombre d'arcs y arrivant
  - Degré=degré sortant+degré entrant
  - Chemin de degré k d'un sommet u à un sommet v : $(u_0,u_1,\ldots,u_k)$ $u=u_0$ et $v=u_k$
  - Chemin élémentaire
  - Circuit et circuit élémentaire
  - Graphe fortement connexe : tout sommet est accessible à partir de tout autre sommet (par un chemin)
  - Composantes fortement connexes d'un graphe : classes d'équivalence de la relation définie comme suit sur l'ensemble des sommets : $R(s_1,s_2)$ si et seulement si il existe un chemin de s_1 vers s_2 et un chemin de s_2 vers s_1

CHAPITRE II
Conception d'algorithmes efficaces

C. Graphes

- **Graphe non orienté :**
  - arêtes
  - **Boucles interdites**
  - Une arête (u,v) est *incidente* aux sommets u et v
  - Degré d'un sommet : nombre d'arêtes qui lui sont incidentes
  - Chaîne et chaîne élémentaire
  - Cycle et cycle élémentaire
  - Graphe acyclique
  - Graphe connexe : chaque paire de sommets est reliée par une chaîne
  - Composantes connexes d'un graphe : classes d'équivalence de la relation définie comme suit sur l'ensemble des sommets : $R(s_1,s_2)$ si et seulement si il existe une chaîne de s_1 vers s_2

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- **Forêt :**
  - graphe non orienté acyclique
- **Arbre :**
  - graphe non orienté acyclique connexe (forêt connexe)

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- G=(S,A) graphe non orienté :
  - G arbre
  - G connexe et |A|=|S|-1
  - G acyclique et |A|=|S|-1
  - Deux sommets quelconques sont reliés par une une unique chaîne élémentaire

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- **Arbre enraciné** (rooted tree) :
  - un sommet se distingue des autres (la racine de l'arbre)
  - La racine d'un arbre enraciné impose un sens de parcours de l'arbre
  - Pour un arbre enraciné : sommets ou nœuds
  - Ancêtre, père, fils, descendant
  - L'unique nœud sans père : la racine
  - Nœuds sans fils : nœuds externes ou feuilles

CHAPITRE II
Conception d'algorithmes efficaces

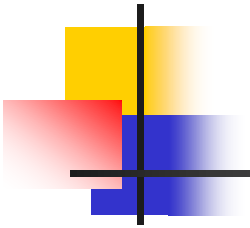
D. Arbres

- Nœuds avec fils : nœuds internes
- Sous-arbre de racine x : l'arbre composé des descendants de x, enraciné en x
- Degré d'un nœud : nombre de fils
- Profondeur d'un nœud x : longueur du chemin entre la racine et le nœud
- Hauteur d'un arbre
- Arbre ordonné

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Arbre binaire (description récursive) :
  - Ne contient aucun nœud (arbre vide)
  - Est formé de trois ensembles disjoints de nœuds : la racine ; le sous-arbre gauche (arbre binaire) ; le sous-arbre droit (arbre binaire)
- Un arbre binaire est plus qu'un arbre ordonné
- Arbre binaire complet
- Arbre n-aire : degré d'un nœud≤n



CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours d'un arbre ordonné
  - Parcours en profondeur d'abord
    - Descendre le plus profondément possible dans l'arbre
    - Une fois une feuille atteinte, remonter pour explorer les branches non encore explorées, en commençant par la branche la plus basse
    - Les fils d'un nœud sont parcourus suivant l'ordre défini sur l'arbre

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours d'un arbre ordonné
  - Parcours en profondeur d'abord

```
PP(A){
   Si Arbre(A) n'est pas réduit à l'arbre vide{
     Pour tous les fils u de racine(A) faire dans l'ordre
                PP(u)
     }
   }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours d'un arbre ordonné
  - Arbre binaire
    - Structure de données récursive permettant sa représentation

```
Type arbre-b{
        valeur : entier;
        sa-gauche : pointeur sur type arbre-b;
        sa-droit : pointeur sur type arbre-b;
        }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours d'un arbre ordonné
  - Arbre binaire
    - Déclarer une variable pointeur sur type arbre-b, qui va être la racine de l'arbre :

Variable racine : pointeur sur type arbre-b;

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours en profondeur d'abord d'un arbre ordonné
  - Arbre binaire

```
PP(A){
   Si A≠NIL{
     PP(A.sa-gauche);
     PP(A.sa-droit);
     }
   }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours en profondeur d'abord d'un arbre ordonné
  - Arbre binaire

```
Préfixe(A){
    Si A≠NIL{
      Afficher(A.valeur);
      Préfixe(A.sa-gauche);
      Préfixe(A.sa-droit);
      }
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours en profondeur d'abord d'un arbre ordonné
  - Arbre binaire

```
Infixe(A){
    Si A≠NIL{
      Infixe(A.sa-gauche);
      Afficher(A.valeur);
      Infixe(A.sa-droit);
      }
    }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

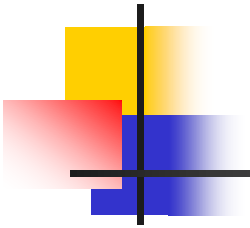
- Parcours en profondeur d'abord d'un arbre ordonné
  - Arbre binaire

```
Postfixe(A){
   Si A≠NIL{
     Postfixe(A.sa-gauche);
     Postfixe(A.sa-droit);
     Afficher(A.valeur);
     }
   }
```

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours en profondeur d'abord d'un arbre ordonné
  - Exemple
    - Utilisation d'un arbre binaire pour le tri d'un tableau par ordre croissant : 50; 40; 45; 60; 30; 55; 44; 100
      - A gauche si ≤; à droite sinon
      - Parcours infixe de l'arbre pour l'affichage du résultat du tri



CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

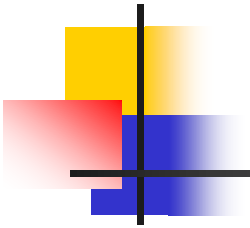
- Parcours d'un arbre ordonné
  - Parcours en largeur d'abord
    - Visiter tous les nœuds de profondeur i avant de passer à la visite des nœuds de profondeur i+1
    - Utilisation d'une file

CHAPITRE II
Conception d'algorithmes efficaces

D. Arbres

- Parcours d'un arbre ordonné
  - Parcours en largeur d'abord

```
PL(A){
    Si A≠NIL{
      F=racine(A);
      Tant que (non FILE-VIDE(F)){
                u=SUPPRESSION(F);          //visiter le noeud
                Pour tous les fils v de u (dans l'ordre) faire
                        INSERTION(F,v);    //faire attendre le noeud
                }
      }
    }
```



CHAPITRE II
Conception d'algorithmes efficaces

E. Parcours d'un graphe

- Initialement, tous les sommets sont blancs
- Lorsqu'un sommet est rencontré pour la première fois, il est colorié en gris
- Lorsque tous les successeurs d'un sommet, dans l'ordre de parcours, ont été visités, le sommet est colorié en noir

CHAPITRE II
Conception d'algorithmes efficaces

E. Parcours d'un graphe

- Parcours en profondeur d'abord

```
PP(G){
    Pour chaque sommet u de G{
            couleur[u]=BLANC;
            }
    Tant que G a des sommets blancs{
            prendre un sommet blanc u;
            VISITER-PP(G,u,couleur);
            }
    }
VISITER-PP(G,s,couleur){
            couleur[s]=GRIS;
            Pour chaque voisin v de s{si couleur[v]=BLANC{VISITER-PP(G,v,couleur);}}
            couleur[s]=NOIR;
            }
```

CHAPITRE II
Conception d'algorithmes efficaces

E. Parcours d'un graphe

- Parcours en largeur d'abord d'un graphe

```
PL(G,s){
    couleur[s]=GRIS;
    Pour chaque sommet u≠s de G{couleur[u]=BLANC;}
    F={s};
    Tant que F≠Φ{
        u=SUPPRESSION(F);
        Tant que u a des voisins blancs{
            prendre un voisin blanc v de u;
            couleur[v]=GRIS;
            INSERTION(F,v);
            }
        couleur[u]=NOIR;
        }
    }
```

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & sinon \end{cases}$$

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & sinon \end{cases}$$

CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

- Récursivité simple
  - La fonction puissance :

$$X^n = \begin{cases} 1 & \text{si } n=0 \\ x*x^{n-1} & \text{sinon} \end{cases}$$

- Récursivité multiple
  - Relation de Pascal donnant les combinaisons C(n,p) :

$$C(n,p) = \begin{cases} 1 & \text{si } p=0 \text{ ou } p=n \\ C(n-1,p)+C(n-1,p-1) & \text{sinon} \end{cases}$$

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & sinon \end{cases}$$

CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

- Récursivité mutuelle

$$Pair(n) = \begin{cases} VRAI & \text{si } n=0 \\ impair(n-1) & \text{sinon} \end{cases}$$

$$impair(n) = \begin{cases} FAUX & \text{si } n=0 \\ pair(n-1) & \text{sinon} \end{cases}$$

- Récursivité imbriquée (la fonction d'Ackermann)

$$A(m,n) = \begin{cases} n+1 & \text{si } m=0 \\ A(m-1,1) & \text{si } m>0 \text{ et } n=0 \\ A(m-1,A(m,n-1)) & \text{sinon} \end{cases}$$

CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

- Exemple : les tours de Hanoï
  - Trois tiges A, B et C sur lesquelles sont enfilés des disques de diamètres tous différents
  - On peut déplacer un seul disque à la fois
  - Il est interdit de poser un disque sur un autre disque plus petit
  - Initialement, tous les disques sont sur la tige A
  - A la fin, tous sont sur la tige C

CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

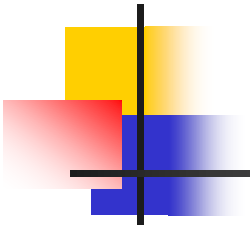
- Exemple : les tours de Hanoï

```
HANOI(n,x,y,z){//x, y et z : départ, intermédiaire, destination
        si n=1 alors déplacer topx vers z
        sinon{   HANOI(n-1,x,z,y);
                 déplacer topx vers z;
                 HANOI(n-1,y,x,z);
                 }
        fin-si
        }
```

CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

- Exemple : les tours de Hanoï
  - Complexité de l'algorithme en nombre de déplacements de disques :
    - $h(n) = \begin{cases} 1 & \text{si } n=1 \\ h(n-1)+1+h(n-1) & \text{sinon} \end{cases}$
    - $h(n) = \begin{cases} 1 & \text{si } n=1 \\ 1+2xh(n-1) & \text{sinon} \end{cases}$
    - $h(n)=2^n-1$
    - $h(n)=O(2^n)=\Theta(2^n)$
  - Complexité exponentielle



CHAPITRE II
Conception d'algorithmes efficaces

F. Itérativité versus récursivité

- Il est toujours possible de dérécursiver un algorithme (récursif), c'est-à-dire de transformer un algorithme récursif en un algorithme itératif équivalent

CHAPITRE II
Conception d'algorithmes efficaces

G. Technique 'diviser pour régner'

- Diviser le problème à résoudre en un certain nombre de sous-problèmes (plus petits)
- Régner sur les sous-problèmes en les résolvant récursivement :
  - Si un sous-problème est élémentaire (indécomposable), le résoudre directement
  - Sinon, le diviser à son tour en sous-problèmes
- Combiner les solutions des sous-problèmes pour construire une solution du problème initial

CHAPITRE II
Conception d'algorithmes efficaces

G. Technique 'diviser pour régner'

- Tri par fusion d'un tableau

```
TRI-FUSION(A,p,r){
    si p<r{
        q=⌊(p+r)/2⌋;
        TRI-FUSION(A,p,q);
        TRI-FUSION(A,q+1,r);
        FUSIONNER(A,p,q,r);
        }
```

CHAPITRE II
Conception d'algorithmes efficaces

G. Technique 'diviser pour régner'

- Tri par fusion d'un tableau

```
FUSIONNER(A,p,q,r){
        i=p;j=q+1;k=1;
        tant que i≤q et j≤r{
                si A[i]<A[j]        alors{C[k]=A[i];i=i+1;}
                                    sinon{C[k]=A[j];j=j+1;}fsi
                k=k+1;
                }
        tant que i≤q{C[k]=A[i];i=i+1;k=k+1;}
        tant que j≤r{C[k]=A[j];j=j+1;k=k+1;}
        Pour k=1 à r-p+1{A[p+k-1]=C[k];}
        }
```

CHAPITRE II
Conception d'algorithmes efficaces

G. Technique 'diviser pour régner'

- Tri par fusion d'un tableau : complexité

$$T(n)=\begin{cases}\Theta(1) \text{ si } n=1\\ 2T(n/2)+\Theta(n) \text{ sinon}\end{cases}$$

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

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Définition : Un arbre binaire de recherche est un arbre binaire tel que pour tout nœud x :

- Tous les nœuds y du sous-arbre gauche de x vérifient clef(y)≤clef(x)
- Tous les nœuds y du sous-arbre droit de x vérifient clef(y)>clef(x)

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Propriété : le parcours (en profondeur d'abord) infixe d'un arbre binaire de recherche permet l'affichage des clés des nœuds par ordre croissant

```
Infixe(A){
    Si A≠NIL{
      Infixe(A.sa-gauche);
      Afficher(A.clef);
      Infixe(A.sa-droit);
      }
    }
```

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Recherche d'un élément :

- la fonction rechercher(A,k) ci-dessous retourne le pointeur sur un nœud dont la clé est k, si un tel nœud existe
- elle retourne le pointeur NIL sinon

```
rechercher(A,k){
    si (A=NIL ou A.clef=k) retourner A
    sinon
            si (k≤A.clef) rechercher(A.sa-gauche,k)
            sinon rechercher(A.sa-droit,k)
    }
```

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Minimum :

- la fonction minimum(A) retourne le pointeur NIL si l'arbre dont la racine est pointée par A est vide
- sinon, elle retourne le pointeur sur un nœud contenant la clé minimale de l'arbre

```
minimum(A){
    si (A=NIL) retourner A
    sinon{
       x=A;
       tant que (x.sa-gauche≠NIL) x=x.sa-gauche;
       retourner x;
       }
    }
```

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Maximum :

- la fonction maximum(A) retourne le pointeur NIL si l'arbre dont la racine est pointée par A est vide
- sinon, elle retourne le pointeur sur un nœud contenant la clé maximale de l'arbre

```
maximum(A){
    si (A=NIL) retourner A
    sinon{
       x=A;
       tant que (x.sa-droit≠NIL) x=x.sa-droit;
       retourner x;
       }
    }
```

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

Insertion d'un élément :

- la fonction insertion(A,k) insère un nœud de clé k dans l'arbre dont la racine est pointée par A

```
Insertion(A,k){
    créer nœud z;z.clef=k;z.sa-gauche=NIL;z.sa-droit=NIL;
    x=A;père_de_x=NIL;
    tant que (x≠NIL){
       père_de_x=x;
       si (k≤x.clef) x=x.sa-gauche
       sinon x=x.sa-droit
       }
    si (père_de_x=NIL) A=z
    sinon              si (k≤père_de_x.clef) père_de_x.sa-gauche=z
                       sinon père_de_x.sa-droit=z
    }
```

CHAPITRE III
Structures de données avancées

A. Arbres binaires de recherche

- Complexité des différentes fonctions :
  - rechercher(A,k), minimum(A), maximum(A), Insertion(A,k)
    - Chacune des fonctions nécessite, dans le pire des cas, une et une seule descente complète (sans retour arrière) dans l'arbre de recherche
    - En d'autres termes, les nœuds visités par chacune des fonctions sont tous sur une unique branche allant de la racine vers une feuille
    - La complexité de chacune des fonctions est donc en O(h), h étant la hauteur de l'arbre de recherche
    - Comme la hauteur est bornée par n-1, la complexité est en O(n)

CHAPITRE IV
NP-complétude

- Problème
  - Description
  - Question
- Instance d'un problème
  - Une instance d'un problème est une spécialisation du problème obtenue en donnant une spécification exacte des données

CHAPITRE IV
NP-complétude

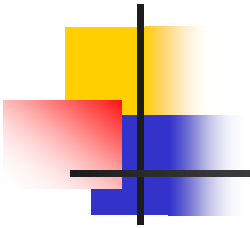
- Le problème SAT
  - Description : une conjonction de m clauses construites à partir de n propositions atomiques
  - Question : la conjonction est-elle satisfiable ?
- Instance du problème SAT
  - La conjonction $p \vee q \wedge p \vee \neg r \wedge \neg p \vee \neg q \vee \neg r$

CHAPITRE IV
NP-complétude

- Le problème PLUS-COURT-CHEMIN
  - Description : un graphe étiqueté et deux sommets u et v du graphe
  - Question : trouver le plus court chemin entre les sommets u et v
- Instance du problème PLUS-COURT-CHEMIN
  - Le graphe étiqueté G=<V,E,c> avec
    - V={A,B,C,D}
    - E={(A,B),(A,C),(A,D),(B,C),(B,D),(C,D)}
    - c(A,B)=2; c(A,C)=3; c(A,D)=5; c(B,C)=3; c(B,D)=1; c(C,D)=1
  - Les sommets A et C

CHAPITRE IV
NP-complétude

- Problème abstrait
  - A un problème donné, on associe un autre problème appelé problème abstrait
  - Un problème abstrait associé à un problème P est défini comme suit :
    - Entrée : une instance du problème P
    - Sortie :
      - une ou plusieurs solutions de l'instance, si solution il y a
      - Inexistence de solution sinon



CHAPITRE IV
NP-complétude

- Problème de décision
  - La théorie de la NP-complétude se restreint aux problèmes dits de décision
  - Un problème de décision est un problème dont chaque instance admet une unique solution, OUI ou NON

CHAPITRE IV
NP-complétude

- Le problème SAT est un problème de décision
  - Description : une conjonction de m clauses construites à partir de n propositions atomiques
  - Question : la conjonction est-elle satisfiable ?
- Instance du problème SAT
  - La conjonction $p \vee q \wedge p \vee \neg r \wedge \neg p \vee \neg q \vee \neg r$

CHAPITRE IV
NP-complétude

- Le problème CHEMIN est un problème de décision
  - Description : un graphe étiqueté, deux sommets u et v du graphe, et un entier k
  - Question : existe-t-il un chemin du graphe reliant u à v, de longueur inférieure ou égale à k ?
- Instance du problème de décision CHEMIN
  - Le graphe étiqueté G=<V,E,c> avec
    - V={A,B,C,D}
    - E={(A,B),(A,C),(A,D),(B,C),(B,D),(C,D)}
    - c(A,B)=2; c(A,C)=3; c(A,D)=5; c(B,C)=3; c(B,D)=1; c(C,D)=1
  - Les sommets A et C
  - L'entier k=2

CHAPITRE IV
NP-complétude

- Problèmes d'optimisation
  - Un problème d'optimisation n'est généralement pas un problème de décision
    - Un problème d'optimisation admet généralement plus d'une solution
  - Mais on peut toujours associer un problème de décision à un problème d'optimisation
- Exemple :
  - Le problème d'optimisation PLUS-COURT-CHEMIN
  - Le problème de décision associé est le problème CHEMIN

CHAPITRE IV : NP-complétude

- Codage
  - Codage d'un ensemble S d'objets abstraits
    - Application e de S dans l'ensemble des chaînes binaires
    - Ou, de façon générale, application de S dans l'ensemble des chaînes d'un alphabet fini
  - Exemple :
    - Codage des entiers sous forme binaire
  - Un algorithme conçu pour résoudre un problème de décision P prend en entrée un codage d'une instance de P

CHAPITRE IV
NP-complétude

- Problème concret
  - Problème dont les instances forment l'ensemble des chaînes d'un alphabet binaire
  - Un algorithme résout un problème concret en O(f(n)) si, pour une instance i de taille n, l'algorithme fournit la solution en O(f(n))
  - En d'autres termes, un algorithme résout un problème concret en O(f(n)) s'il existe un entier $n_0 \geq 1$ et une constante $c \geq 0$ tels que :
    - pour toute instance i de taille $n \geq n_0$, le nombre T(n) d'opérations élémentaires nécessaires à l'algorithme pour fournir la solution de l'instance est borné par c*T(n) : $T(n) \leq c*f(n)$

CHAPITRE IV
NP-complétude

- La classe de complexité P
  - La classe de complexité P est l'ensemble des problèmes concrets de décision qui sont résolubles en un temps polynomial
  - Un problème concret est résoluble en un temps polynomial s'il existe un algorithme permettant de le résoudre en $O(n^k)$, pour une certaine constante k

CHAPITRE IV
NP-complétude

- Fonction calculable en temps polynomial
  - Une fonction $f:\{0,1\}^* \rightarrow \{0,1\}^*$ est calculable en temps polynomial s'il existe un algorithme polynomial la calculant, c'est-à-dire produisant f(x), pour tout $x \epsilon \{0,1\}^*$.
  - Deux codages e_1 et e_2 définis sur un même ensemble S sont reliés polynomialement s'il existe deux fonctions calculables en temps polynomial, f_{12} et f_{21}, telles que pour tout $s \epsilon S$, $f_{12}(e_1(s))=e_2(s)$ et $f_{21}(e_2(s))=e_1(s)$

CHAPITRE IV
NP-complétude

- La classe de complexité NP
  - Algorithme de validation :
    - Considérons l'instance suivante de SAT :

      $p \vee q \wedge p \vee \neg r \wedge \neg p \vee \neg q \vee \neg r$

    - Si on considère une instanciation (b_1,b_2,b_3) du triplet (p,q,r) de propositions atomiques, on peut facilement vérifier si oui ou non elle satisfait l'instance. Et si elle la satisfait, l'instanciation peut être vue comme un certificat que l'instance du problème SAT est satisfiable
    - Soit Q un problème concret de décision. Un algorithme de validation pour Q est un algorithme de décision A à deux arguments, une instance du problème Q, et un certificat y. L'algorithme A valide l'instance x si et seulement si il existe un certificat y tel que A(x,y)=vrai :
      - A valide une instance x de Q si et seulement si la solution de x est OUI

CHAPITRE IV
NP-complétude

- La classe de complexité NP
  - La classe de complexité NP est l'ensemble des problèmes concrets de décision pour lesquels il existe un algorithme polynomial de validation
  - NP : Non-déterministe Polynomial
- Réductibilité :
  - Soient Q_1 et Q_2 deux problèmes concrets. Q_1 est réductible en temps polynomial à Q_2 ($Q_1 \leq_P Q_2$) s'il existe une fonction calculable en temps polynomial $f:\{0,1\}^* \rightarrow \{0,1\}^*$ telle que pour tout $x \in \{0,1\}^*$:
    - La solution de l'instance x de Q_1 est OUI

      si et seulement si
    - la solution de l'instance f(x) de Q_2 est OUI

CHAPITRE IV
NP-complétude

- Problème NP-complet

Un problème Q est NP-complet si

1. $Q \in NP$
2. $\forall Q' \in NP,\ Q' \leq_P Q$

- *On note NPC la classe des problèmes NP-complets*
- *Un problème concret qui vérifie la propriété 2 mais pas nécessairement la propriété 1 est dit* ***NP-difficile***

CHAPITRE IV
NP-complétude

- Exemples de problèmes NP-complets
  - Le problème SAT
  - Le problème 3-SAT
  - CYCLE HAMILTONIEN :
    - Existence dans un graphe d'un cycle hamiltonien
  - VOYAGEUR DE COMMERCE :
    - Existence dans un graphe étiqueté d'un cycle hamiltonien de coût minimal
  - CLIQUE :
    - Existence dans un graphe d'une clique (sous-graphe complet) de taille k
  - 3-COLORIAGE D'UN GRAPHE :
  - PARTITION :
    - Peut-on partitionner un ensemble d'entiers en deux sous-ensembles de même somme ?

CHAPITRE IV
NP-complétude

- Preuve de NP-complétude
  - Comment démontrer qu'un problème est NP-complet ?

Théorème

- Si Q_1 est un problème tel que $Q_2 \leq_P Q_1$ pour un certain problème $Q_2 \in$ NPC, alors Q_1 est NP-difficile
- Si, de plus, $Q_1 \in$ NP, alors $Q_1 \in$ NPC.

CHAPITRE IV
NP-complétude

- Méthode pour montrer la NP-complétude d'un problème Q_1
  - Prouver que $Q_1 \in NP$
  - Choisir un problème NP-complet Q_2
  - Décrire un algorithme polynomial capable de calculer une fonction f faisant correspondre toute instance de Q_2 à une instance de Q_1
  - Démontrer que la fonction f satisfait la propriété :

    $Q_2(x)$=oui si et seulement si $Q_1(f(x))$ = oui
  - Démontrer que l'algorithme calculant f s'exécute en temps polynomial

CHAPITRE IV
NP-complétude

- Montrer que le problème de 3-coloriage d'un graphe est NP-complet ?