LICENCE DE MATHEMATIQUES **1999**
C4 - INFORMATIQUE

SUPPORT DE COURS

Claudine Chaouiya Chantegrel

1. Introduction à l'analyse d'algorithmes

1.1 Notion d'algorithme

Un **algorithme** est une "suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème" (le Petit Larousse Illustré, 1994).

On a tous des dizaines d'exemples d'algorithmes en tête :

l'algorithme d'Euclide qui permet de calculer le pgcd de deux nombres entiers,
l'algorithme de multiplication décimale
l'algorithme de recherche dans une suite ordonnée (dictionnaire)
...

Lorsqu'on écrit un programme, on met en oeuvre un algorithme, c'est pourquoi cette notion est fondamentale en informatique. Mais attention, on ne peut réduire la notion d'algorithme à celle de méthodes informatiques de résolution de problèmes.

Etymologie et petit historique

Al Khuwarismi, mathématicien perse, né en 783, et qui a *Maison de la Sagesse* à Bagdad, a donné le nom de l'un de ses ouvrages à une branche fondamentale des mathématiques, l'algèbre, et son propre nom à la science dite *algorithmique*. Mais on peut faire remonter la pratique algorithmique bien avant, avec les savants babyloniens de 1800 avant J.C. !

Il est vrai que la notion d'algorithme est aujourd'hui très liée à l'idée d'ordinateur.
Vers 1930, plusieurs mathématiciens (Gödel, Church, Turing, Post) ont formalisé le concept d'algorithme – ou calcul – par différents modèles abstraits. La théorie de la "calculabilité" s'intéresse à l'existence d'un algorithme pour la résolution d'un problème donné. C'est une théorie importante mais qui ne fait pas l'objet de notre cours.

Exemple fondamental de problème non décidable (non calculable) :

il ne peut exister un programme universel U, qui soit capable de répondre, en un nombre fini d'opérations, à la question suivante, relativement à un programme P quelconque : *le programme P s'arrêtera-t-il au bout d'un nombre fini d'opérations ?i* Car si un tel programme existait, on aboutirait à un contradiction logique en l'appliquant simplement à lui-même (Histoire universelle des chiffres de G.Ifrah).

Nous nous intéressons à l'analyse d'algorithmes et non au problème de leur existence. Il ne suffit de connaître un algorithme, encore faut-il pouvoir estimer son efficacité.
Objectifs : - estimer l'efficacité **intrinsèque** d'algorithmes, indépendamment de la machine, du langage, et autres détails d'implémentation

- comparer des algorithmes

La **complexité d'un algorithme** est le temps et l'espace mémoire nécessaires à son exécution.
Modèle d'ordinateur de référence : une unité de calcul et une mémoire telles que a) l'accès (et le rangement) d'un élément dans la mémoire se fait en un temps fixe, b) l'unité de calcul ne peut traiter qu'une opération à la fois.

1.2 Mesure de la complexité

On peut toujours identifier des opérations élémentaires : le temps d'exécution de l'algorithme sera proportionnel au nombre de ces opérations

Exemples :

1. recherche d'un élément dans une liste → nombre de comparaisons
2. multiplication de matrices → nombre de multiplications et d'additions
3. tri d'une liste d'éléments → nombre de comparaisons et de déplacements

En faisant varier le nombre d'opérations élémentaires considérées, on fait varier le degré de précision de l'analyse et son degré d'abstraction (degré d'indépendance par rapport à l'implémentation)
Le temps d'exécution est supposé proportionnel à la mesure choisie, on ne peut comparer que des algorithmes qui utilisent les mêmes opérations

Pour calculer la complexité en temps, on compte les opérations élémentaires de l'algorithme que l'on analyse. Il n'y a pas de méthode complète pour faire cela, mais il existe quelques règles:

- séquence d'instructions : ajouter
- branchement conditionnel : majorer
- boucle : S P(i) où P(i) est le nombre d'opérations au $i^{ème}$ passage dans la boucle (i variable de contrôle de la boucle)
- appel de procédure/fonction : nombre d'opérations de la procédure/fonction
- pour les procédures ou fonctions récursives : résoudre des relations de récurrence T(n)= f(T(k)) où k<n

Exemple : la fonction récursive "factorielle"

```
function fact (n : integer) : integer ;
begin
    if n = 0 then fact := 1
    else fact := n * fact (n-1)
end ;
```

on peut choisir comme opération élémentaire la multiplication de 2 entiers. Alors on a

$$T(0) = 0 \text{ et } T(n) = T(n-1) + 1 \text{ pour } n \geq 1$$

que l'on résout facilement : T(n) = n.

Exemple : un algorithme de recherche

Algorithme de recherche séquentielle

```
                type liste = array[1..n] of integer;
                function trouve(L:liste; X:integer): integer;
                var          j : integer;
                begin
1                       j:=1;
2                       while (j<=n) and (L[j]<>X)
3                            do j := j+1;
4                       if j>n then j:=0;
                        trouve:=j;
                end;
```

Il s'agit de compter le nombre d'itérations et, par itération, le nombre de comparaisons.

Les instructions j<=n(**2**) et j := j+1(**3**) ne sont pas prises en compte car elles dépendent de la programmation et de la structure de données.
Les opérations significatives sont donc les comparaisons L[j]<>X il y en a une par itération.
Le nombre d'itérations est n si $X \notin L$, j (rang de la 1ère occurrence) sinon.

L'analyse se fait en déterminant des **invariants de boucle** (propriétés vraies à chaque itération) et des **conditions d'arrêt.**

invariants de boucle :	au début de la 1ère itération, j=1
	au début de la kième itération, j=k et $\forall i, 1 \leq i < k, L[i] \neq X$
conditions d'arrêt :	si au début de la kième itération on a k≤n et L[k]=X, le programme s'arrête avec j=k
	si on a k=n+1, le programme s'arrête avec j=0.

1.3 Autres critères d'analyse

- place mémoire: très souvent la rapidité d'un algorithme est au détriment de l'espace utilisé (compromis espace-temps)
- simplicité de l'algorithme: implémentation et maintenance (temps "humain")
- adéquation aux données: exemple de l'algorithme de tri par insertion, mauvais en général, performant sur des listes presque triées

Exemple du tri par insertion :

Algorithme du tri par insertion

```
type liste = array[1..n] of integer;
procedure tri_insertion(A : liste);
var cle,i,j : integer;
begin
    for i:=2 to longueur(A) do
    begin
        cle := A[i];
        j := i-1;
        while (j>0) and (A[j]>cle) do
        begin
            A[j+1] := A[j];
            j := j-1;
        end;
        A[j+1] := c;
    end;
end;
```

1.4 Complexité en moyenne et au pire

Le temps d'exécution d'un algorithme dépend des entrées sur lesquelles il est exécuté (cf. algorithme de la recherche séquentielle)
Il faut définir la taille des entrées (en général le nombre d'éléments manipulés).
Pour une taille fixée des entrées, la complexité de certains algorithmes varie en fonction des entrées elles-mêmes.

Définitions :

Soient D_n l'ensemble des entrées de taille n et $C_A(d)$ la complexité en temps de l'algorithme A sur l'entrée d, alors :

- la complexité dans le **meilleur des cas** est définie par :
$$Min_A(n) = \min\{C_A(d),\ d \in D_n\}$$
- la complexité dans le **pire des cas** est définie par :
$$Max_A(n) = \max\{C_A(d),\ d \in D_n\}$$
- la complexité **en moyenne** est définie par :
$$Moy_A(n) = \sum_{d \in D_n} p(d).C_A(d)$$

où p(d) est la probabilité associée à l'entrée d.

Remarques :

1. les complexités dans le pire et le meilleur des cas donnent des bornes sur la complexité de l'algorithme sur des entrées de taille n. En général on s'intéresse à la complexité dans le pire des cas.

2. si toutes les entrées sont équiprobables, alors la complexité en moyenne s'exprime simplement en fonction du cardinal $|D_n|$:

$$Moy_A(n) = \frac{1}{|D_n|} \sum_{d \in D_n} C_A(d)$$

3. en général les entrées ne sont pas équiprobables et il faut définir un modèle probabiliste du problème. La complexité en moyenne est souvent difficile à déterminer, pour beaucoup d'algorithmes elle n'est pas connue.
4. on a : $Min_A(n) \leq Moy_A(n) \leq Max_A(n)$, $\forall n$. Si le comportement de A ne dépend que de la taille de l'entrée, ces 3 quantités sont confondues.

Exemple :

Algorithme de multiplication de matrices carrées (C=AB)

```
type matrice = array[1..n,1..n] of integer;
var i,j,k : integer;
begin
    for i:=1 to n do
        for j:=1 to n do
        begin
            C[i,j]:=0;
            for k:=1 to n do
                C[i,j]:=C[i,j] + A[i,k] * B[k,j]
        end;
end;
```

opération élémentaires : les multiplications,

on a $Min(n) = Moy(n) = Max(n) = \sum_i^n \sum_i^n \sum_i^n 1 = n^3$

Exemple : algorithme de recherche séquentielle

Opérations élémentaires : les comparaisons.

On a déjà vu que Max(n) = n et Min(n) = 1, reste à calculer Moy(n), sachant que :
- la probabilité pour que l'élément cherché X soit dans la liste L est q
- si $X \in L$, toutes les places sont équiprobables

Soit $D_{n,i}$, $1 \leq i \leq n$, l'ensemble des données où X est à la ième place ; on a $p(D_{n,i}) = \frac{q}{n}$

Soit $D_{n,0}$, $1 \leq i \leq n$, l'ensemble des données où X est absent; on a $p(D_{n,0}) = 1-q$

Il est clair que $coût(D_{n,i})=i$ et $coût(D_{n,0})= n$, d'où

$$Moy(n) = \sum_{i=0}^{n} p(D_{n,i})\, coût(D_{n,i}) = (1-q)n + \frac{q}{n} \sum_{1}^{n} i = (1-q)n + \frac{q(n+1)}{2}$$

Si q=1 ($X \in L$) on a $Moy(n) = \frac{n+1}{2}$,

si q=1/2 (une chance sur 2 pour que $X \in L$) on a $Moy(n) = \frac{3n+1}{4}$

2. Notations asymptotiques

La complexité d'un algorithme est une fonction de la taille de l'entrée. Il est important de connaître le comportement asymptotique de cette fonction pour pouvoir comparer des algorithmes sur des problèmes de taille élevée.
On s'intéresse à l'ordre de grandeur asymptotique de la fonction de complexité.
Par exemple,

- l'algorithme A de complexité 25n est meilleur que l'algorithme B de complexité $3n^2$

pour tous k_1, k_2 (>0), l'algorithme A de complexité $k_1 n$ est meilleur que l'algorithme B de complexité $k_2 n^2$ car la fonction $f(n)=n^2$ croît plus vite que la fonction $g(n)=n$

2.1 Définition de la notation θ

Pour une fonction donnée g, on note θ(g(n)) l'ensemble de fonctions :

$\theta(g(n)) = \{f(n) / \exists\ c_1, c_2 >0, \exists\ n_0, 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \forall\ n \geq n_0\ \}$

on écrit $f(n)=\theta(g(n))$ pour $f(n) \in \theta(g(n))$ et on dit que g(n) est une **borne approchée asymptotique** pour f(n).

NB: toute fonction utilisée dans la notation θ est supposée positive asymptotiquement.

2.2 Définition de la notation O

La notation θ borne une fonction asymptotiquement à la fois par excès et par défaut. Lorsqu'on ne dispose que d'une borne supérieure asymptotique, on utilise la notation O.

Pour une fonction donnée g(n) on note O(g(n)) l'ensemble de fonctions :
$O(g(n)) = \{f(n) / \exists\ c >0, \exists\ n_0 >0, 0 \leq f(n) \leq c\ g(n), \forall\ n \geq n_0\ \}$

on écrit $f(n)=O(g(n))$ pour $f(n) \in O(g(n))$ et on dit que g(n) est une **borne supérieure asymptotique** pour f(n)

2.3 Définition de la notation Ω

Pour une fonction g(n) donnée, on note Ω(g(n)) l'ensemble des fonctions:
$\Omega(g(n)) = \{f(n) / \exists\ c >0, \exists\ n_0 >0, 0 \leq c\ g(n) \leq f(n), \forall\ n \geq n_0\ \}$

On écrit $f(n)=\Omega(g(n))$ pour $f(n) \in \Omega(g(n))$

3. Récurrences

Souvent, le temps d'exécution d'un algorithme récursif s'écrit comme une récurrence (équation ou inégalité qui décrit une fonction à partir de sa valeur sur des entrées plus petites).
Autrement dit la mesure sur une taille n T(n) est fonction de la mesure sur des tailles inférieures : $T(n) = f(\{T(p), p<n\})$.

3.1 Classes d'équations de récurrences

1. récurrences linéaires d'ordre k : $T(n) = f(n, T(n-1), ..., T(n-k)) + g(n)$
avec, $k \geq 1$ entier fixé
f combinaison linéaire des T(i), pour i = n-k,..,n-1,
g fonction quelconque de n;

2. récurrences de partitions : $T(n) = a\,T(n/b) + d(n)$
avec, a et b constantes entières
d(n) fonction quelconque de n.

3. récurrences complètes, linéaires ou polynomiales : $T(n) = f(n,T(n-1), ..., T(0))+g(n)$
avec, f fonction linéaire ou polynomiale des T(i), i=1,...,n
g(n) une fonction quelconque de n.

3.2 Quelques techniques de résolution

3.2.1 Récurrences linéaires, méthode des facteurs sommants

On écrit la relation à l'ordre n, n-1, n-2, ..., 1. On multiplie par un facteur adapté pour pouvoir sommer et simplifier.

Exemple : $T(n) = T(n-1) + 2^n$, $T(0) = 1$

3.2.2 Récurrences linéaires à coefficients constants, équation caractéristique

a) récurrence linéaire **homogène**, d'ordre k :

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + ... + a_kT(n-k) = 0, \text{ avec } a_0, a_1,...,a_k \text{ constantes}$$

la résolution d'une telle équation passe par celle de **l'équation caractéristique** :

$$a_0x^k + a_1x^{k-1} + ... + a_k = 0$$

on cherche les racines de cette équation : $r_1, r_2, ..., r_p$ de multiplicité respective $m_1, m_2, ...,m_p$.
La solution s'écrit :

$$T(n) = \sum_{i=1}^{p}\left(\sum_{j=0}^{m_i-1} c_{ij}\, n^j\, r_i^n\right)$$, les constantes c_{ij} étant déterminées par les conditions initiales.

Exemple : $T(n) - 3T(n-1) - 4T(n-2) = 0$ pour $n \geq 2$, $T(0) = 0$ et $T(1) = 1$

b) récurrence linéaire **non homogène**, d'ordre k de la forme :

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + ... + a_kT(n-k) = b_1^n P_1(n) + b_2^n P_2(n) + ...+ b_q^n P_q(n)$$

avec a_0, a_1,...,a_k et b_1, b_2,..., b_q constantes et $P_1(n)$, $P_2(n)$,...,$P_q(n)$ polynômes en n de degré d_1, d_2, ..., d_q respectivement.
L'équation caractéristique s'écrit alors :

$$(a_0x^k + a_1x^{k-1} + ... + a_k) (x-b_1)^{d_1+1} (x-b_2)^{d_2+1} ... (x-b_q)^{d_q+1}= 0.$$

Comme précédemment, on cherche les racines de cette équation (avec leur multiplicité respective) et on détermine la solution générale, les facteurs constants étant déterminés par les conditions initiales.

Exemple : $T(n) = 2T(n-1) + n + 2$, $n \geq 1$, $T(0) = 0$

3.2.3 Récurrences de partition

On se restreindra aux récurrence de partition de la forme :

$$T(n) = a.T\left(\frac{n}{b}\right) + d(n),\ n \geq 2,\ a \text{ et } b \text{ constantes}$$
$$T(1) = 1$$

On suppose que n est une puissance de b : $n = b^k$
Dans l'équation initiale, on remplace n par b^k : $T(b^k) = a.T(b^{k-1}) + d(b^k)$
Posons $t_k = T(b^k)$, on obtient alors une équation linéaire : $t_k = a.t_{k-1} + d(b^k)$
$t_0 = 1$
On résout cette équation par la méthode des facteurs sommants :

$$\begin{array}{rrcr} t_k = & a.t_{k-1} & + & d(b^k) \\ a.t_{k-1} = & a^2.t_{k-2} & + & a.d(b^{k-1}) \\ a^2.t_{k-2} = & a^3.t_{k-3} & + & a^2.d(b^{k-2}) \\ ..... & ..... & & \\ a^{k-1}.t_1 = & a^k.t_0 & + & a^{k-1}.d(b^0) \end{array}$$

$$t_k = a^k + \sum_{i=0}^{k-1} a^i.d(b^{k-i})$$

Or $k = \log_b(n)$ donc $a^k = a^{\log_b(n)} = n^{\log_b(a)}$ donc,

Eq.(A) $$\mathbf{T(n) = n^{\log_b(a)} + \sum_{i=0}^{k-1} a^i.d\left(\frac{n}{b^i}\right)}$$

<u>Ordre de grandeur dans le cas où d est une fonction multiplicative :</u>
Une fonction d est dite multiplicative si d(u.v) = d(u).d(v). Par exemple, $d(n) = n^p$ est multiplicative, d(n) = n-1 ne l'est pas.

Si on suppose d multiplicative, on a : $\sum_{i=0}^{k-1} a^i.d(b^{k-i})) = d(b)^k \sum_{i=0}^{k-1} \left(\frac{a}{d(b)}\right)^i$

- Si a = d(b), le second terme vaut : $k.d(b)^k = \log_b(n).n^{\log_b(d(b))}$ car $k = \log_b(n)$
- Si a ≠ d(b), le second terme s'écrit : $d(b)^k.\frac{1-\left(\frac{a}{d(b)}\right)^k}{1-\frac{a}{d(b)}} = \frac{n^{\log_b(a)} - n^{\log_b(d(b))}}{\frac{a}{d(b)} - 1}$

Finalement, on obtient

1. si a = d(b) alors $T(n) = \theta(n^{\log_b(a)}.\log_b(n))$

dans le cas où $d(n) = n^\alpha$, on a $T(n) = \theta(n^\alpha.\log_b(n))$

2. si a > d(b) alors $T(n) = \theta(n^{\log_b(a)})$

3. si a < d(b) alors $T(n) = \theta(n^{\log_b(d(b))})$

dans le cas où $d(n) = n^\alpha$, on a $T(n) = \theta(n^\alpha)$

<u>Exemples :</u>

$T(n) = 2.T(n/2) + n - 1, \quad n \geq 2$
$T(1) = 1$

Eq (A) donne (a = b = 2) $\quad T(n) = n + \sum_{i=0}^{k-1} 2^i \left(\frac{n}{2^i} - 1\right) = n + k.n - (2^k - 1)$

comme $k = \log_2 n$, finalement $T(n) = n.\log_2 n + 1$

$T(n) = 3.T(n/2) + c.n$ $\quad$ n>1 et n puissance de 2, c constante
$T(1) = 1$

Eq (A) donne (a = 3, b = 2) $\quad T(n) = n^{\log_2(3)} + \sum_{i=0}^{k-1} 3^i.c.\left(\frac{n}{2^i}\right) =$

$n^{\log_2(3)} + c.n.\frac{(3/2)^k - 1}{3/2 - 1} =$

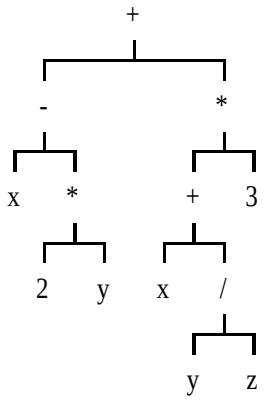
$n^{\log_2(3)} + 2.c.(n - n^{\log_2(3)}) = \theta(n^{\log_2(3)})$

4. Les arbres binaires

Un arbre binaire est soit vide (noté Ø) soit de la forme B = <o, B_1, B_2> où B_1, B_2 sont des arbres binaires disjoints et o est un noeud appelé racine.

Exemple :

expression arithmétique (x-(2*y)) + ((x+(y/z)) *3)



La structure d'arbre binaire est utilisée dans de très nombreuses applications informatiques.

Il est important de noter la non-symétrie gauche-droite des arbres binaires : l'arbre <o, <o, Ø, Ø>, Ø> et l'arbre <o, Ø, <o, Ø Ø>> sont des arbres binaires différents

Opérations de base sur les arbres binaires:
L'opération *arbre-vide*(A) rend vrai si A= Ø, faux sinon
L'opération *racine*(A) est définie si A≠ Ø ; si A = < o, B_1 , B_2 >) elle rend noeud o (la racine de l'arbre)
L'opération *g*(A) est définie si A≠ Ø ; si A = < o, B_1 , B_2 >) elle rend B_1, l'arbre gauche de A
L'opération *d*(A) est définie si A≠ Ø ; si A = < o, B_1 , B_2 >) elle rend B_2, l'arbre droit de A

L'opération *contenu* associe une information de type Elément à chaque Noeud de l'arbre. Un Arbre dont les noeuds contiennent des informations est dit **étiqueté.** Si B= < o , B_1 , B_2 > est un arbre étiqueté dont la racine contient l'élément *r* , on note abusivement B = < *r* , B_1 , B_2 >

4.1 Terminologie

Soit B= < o , B_1 , B_2 > un arbre binaire,

- o est la racine de B
- B_1 est le sous-arbre de gauche de la racine de B et B_2 est le sous-arbre de droite de la racine de B
- C est un sous-arbre de B ssi C = B, B_1 ou B_2 ,ou C est un ss-arbre de B_1 ou B_2
- on appelle fils gauche (resp. fils droit) d'un noeud, la racine de son sous-arbre de gauche (resp. de droite)
- si un noeud n_i a pour fils gauche (resp.droit) n_j , on dit que n_i est le père de n_j
- deux noeuds qui ont le même père sont dits frères
- le noeud n_i est un ascendant (ancêtre) de n_j ssi n_i est le père de n_j ou un ascendant du père de n_j
- n_i est un descendant de n_j ssi n_i est le fils de n_j ou un descendant d'un fils de n_j

- tous les noeuds d'un arbre binaire ont au plus 2 fils :
  un noeud qui a 2 fils est un noeud interne (ou point double)
  un noeud qui a seulement un fils gauche (resp.droit) est un point simple à gauche (resp. à droite), ou noeud interne sens large
  un noeud sans fils est appelé une feuille (noeud externe)
- on appelle branche de B tout chemin (toute suite de noeuds consécutifs) de la racine à une feuille de B

4.2 Mesures

- la taille d'un arbre est le nombre de ses noeuds :

  $taille\ (\varnothing) = 0$
  $taille\ (< o , B_1 , B_2 >) = 1 + taille(B_1) + taille(B_2)$

- la hauteur d'un noeud x (profondeur ou niveau) est définie comme suit :

  $h\ (x) = 0$ si x est la racine de B
  $h\ (x) = 1 + h\ (y)$ si y est le père de x

- la hauteur d'un arbre B est :

  $h\ (B) = \max\{h(x), x \text{ noeud de } B\}$

- la longueur de cheminement d'un arbre B est :

  $$LC\ (B) = \sum_{x \text{ noeud de } B} h\ (x)$$

- dans la longueur de cheminement on peut distinguer la contribution des feuilles de celle des autres noeuds :

  la longueur de cheminement externe d'un arbre *LCE* est la somme des hauteurs prise sur toutes les feuilles

  la longueur de cheminement interne d'un arbre *LCI* est la somme des hauteurs prise sur tous les noeuds internes
  on a $LC\ (B) = LCE\ (B) + LCI\ (B)$

Avec ces quantités on peut en déterminer d'autres, par exemple la profondeur moyenne externe d'un arbre ayant *f* feuilles est :

$$PE\ (B) = \frac{1}{f}\ LCE\ (B)$$

4.3 Arbres binaires particuliers

- un arbre binaire dégénéré (ou filiforme) est un arbre formé uniquement de points simples
- un arbre binaire est dit complet s'il contient 1 noeud au niveau 0, 2 au niveau 1, 4 au niveau 2, ..., 2^h au niveau h
  quel est le nombre total de noeuds d'un arbre complet de hauteur h ?

- un arbre binaire parfait est un arbre dont tous les niveaux sont complètement remplis sauf éventuellement le dernier, et dans ce cas les feuilles du dernier niveau sont regroupées le plus à gauche possible
- un arbre binaire localement complet est un arbre binaire non vide qui n'a pas de point simple (tous les noeuds qui ne sont pas des feuilles ont 2 fils)
- un peigne gauche (resp. peigne droit) est un arbre binaire localement complet dans lequel tout fils droit (resp. gauche) est une feuille

4.4 Occurrence et numérotation hiérarchique

Pour désigner un noeud dans un arbre, on lui associe un mot formé de symboles '0' et '1' décrivant le chemin de la racine à ce noeud : c'est l'occurrence du noeud dans l'arbre. Occurrence de la racine est le mot vide ε et si un noeud a pour occurrence μ son fils gauche a pour occurrence μ0 et son fils droit μ1.

Dans la numérotation en ordre hiérarchique des noeuds d'un arbre complet, on numérote en ordre croissant à partir de 1 tous les noeuds à partir de la racine, niveau par niveau et de gauche à droite sur chaque niveau.
Un noeud numéroté i a son fils gauche numéroté par $2i$ et son fils droit par $2i+1$.

En déduire que si un noeud d'un arbre complet a pour occurrence μ et pour numérotation en ordre hiérarchique i, on a la relation :

$i = 2^{\lfloor \log_2 i \rfloor} + m$ où m est l'entier représenté par μ.

4.5 Propriétés fondamentales :

La profondeur et la longueur de cheminement sont des mesures fondamentales pour l'analyse de la complexité d'un certain nombre d'algorithmes.

Encadrement de la hauteur et de la longueur de cheminement

1 - pour un arbre binaire ayant n noeuds et de hauteur h : $\lfloor \log_2 n \rfloor \leq h \leq n-1$

tout arbre binaire non vide B ayant f feuilles a une hauteur h (B)$\geq \lceil \log_2 f \rceil$

2 - pour un arbre binaire B ayant n noeuds au total, on a : $\sum_{k=1}^{n} \lfloor \log_2 k \rfloor \leq LC(B) \leq \frac{n(n-1)}{2}$

La longueur de cheminement d'un arbre binaire ayant n noeuds est au minimum de l'ordre de $n\log_2 n$ et au maximum de l'ordre de n^2

3 - tout arbre binaire B ayant f feuilles a une profondeur moyenne externe : $PE(B) \geq \log_2 f$

Propriétés des arbres binaires localement complets

1 - un arbre binaire localement complet ayant n noeuds internes a $(n+1)$ feuilles

2 - B un arbre binaire localement complet ayant n noeuds internes : $LCE(B) = LCI(B) + 2n$

On appelle complétion locale d'un arbre binaire B l'opération qui consiste à compléter l'arbre B en un arbre Binaire Complet, en rajoutant des feuilles de telle sorte que chaque noeud de B ait 2 fils dans BC

Dénombrement des arbres binaires

1 - le nombre d'arbres binaires de taille n est $b_n = \frac{1}{n+1} C_{2n}^{\ n}$

2 - le nombre bc_n d'arbres binaires localement complets ayant $(2n+1)$ noeuds est

$bc_n = \frac{1}{n+1} C_{2n}^{\ n}$

4.6 Représentation des arbres binaires

Utilisation de pointeurs

à chaque noeud, on associe 2 pointeurs, un vers le sous-arbre gauche, l'autre vers le sous-arbre droit, l'arbre est déterminé par l'adresse de sa racine. Quand l'arbre est étiqueté, l'information des noeuds est mise dans un champ supplémentaire.

```
type    ARBRE    =↑Noeud
        Noeud    = enregistrement
                       val : Element
                       g,d : ARBRE
                   fin
```

Utilisation de tableaux

C'est une simulation de la représentation précédente. A chaque noeud on associe un indice dans un tableau à deux champs (G et D). Le champ G (resp. D) contient l'indice associé à la racine du sous-arbre gauche (resp. droit).

Il faut une convention pour l'indice associé à l'arbre vide. Il faut aussi connaître l'indice dans le tableau de la racine. De plus lorsque l'arbre est étiqueté, on représente l'information associée dans un champ supplémentaire.

```
type    TAB = tableau [1..N] de    enregistrement
                                       V : Element
                                       G : 0..N
                                       D : 0..N
                                   fin
  ARBTAB = enregistrement
                   rac : 0..N ;
                   tab : TAB;
               fin
```

Le cas des arbres parfaits - représentation séquentielle

La numérotation en ordre hiérarchique permet de coder les arbres parfaits de taille n dans un tableau de n cases et on a,

$2 \leq i \leq n \Rightarrow$ le père du noeud d'indice i est à l'indice (i div 2)
$1 \leq i \leq (n \text{ div } 2) \Rightarrow$ le fils gauche du noeud d'indice i est en $2i$ et le fils droit du noeud d'indice i est en $2i+1$

Cette représentation peut être utilisée pour des arbres binaires quelconques, en laissant des cases vides dans le tableau, qui correspondent aux noeuds non présents. Elle perd alors l'avantage de sa compacité : un arbre de n noeud peut nécessiter jusqu'à $2^n - 1$ cases.

4.7 Parcours en profondeur à main gauche d'un arbre binaire

Le parcours d'un arbre c'est l'examen systématique, dans un certain ordre, de chacun des noeuds de l'arbre.

Le chemin part à gauche de la racine et va toujours le plus à gauche possible en suivant l'arbre. Dans ce parcours, chaque noeud est rencontré trois fois (descente, remontée gauche, remontée droite).

Algorithme récursif :

```
procedure  Parcours(A : Arbre);
debut
        si A = arbre-vide alors TERMINAISON
        sinon debut
                                TRAITEMENT1
                                Parcours(g(A))
                                TRAITEMENT2
                                Parcours(d(A))
                                TRAITEMENT3
                fin
fin
```

La complexité comptée en nombre de traitements de noeuds sur un arbre de *n* noeuds est en $\theta(n)$.
Le parcours en profondeur à main gauche contient comme cas particuliers 3 ordres classiques d'exploration d'arbres :

1. **ordre préfixe** ou **préordre** : si TRAITEMENT2 et TRAITE-MENT3 n'existent pas, alors TRAITEMENT1 est appliqué en préordre (n_1,n_2,n_4,n_8,n_{11},n_9,n_5,n_3,n_6,n_{10},n_7)

2. **ordre infixe** ou **symétrique** : seul TRAITEMENT2 est appliqué
(n_8,n_{11},n_4,n_9,n_2,n_5,n_1,n_{10},n_6,n_3,n_7)

3. **ordre suffixe** ou **postfixe** : seul TRAITEMENT3 est appliqué
(n_{11},n_8,n_9,n_4,n_5,n_2,n_{10},n_6,n_7,n_3,n_1)

Remarque : on ne peut pas, avec ce parcours, obtenir l'ordre hiérarchique, car il correspond à un parcours par niveaux et non à un parcours en profondeur.

Dérécursification du parcours d'arbres binaires :

Principe : descendre la branche gauche de l'arbre en effectuant T1 et en empilant les noeuds rencontrés; on effectuera T2 avec ces noeuds lors de la remontée gauche. L'ordre de rencontre des noeuds pour T2 est inverse de l'ordre pour T1, d'où la structure de pile. Quand un noeud est rencontré en remontée gauche, on effectue T2, on mémorise le noeud dans une pile, puis on travaille sur tout son sous-arbre droit pour enfin traiter le noeud en remontée droite par T3.

Pour éviter d'utiliser 2 piles, une marque N est utilisée qui indique le sens de la remontée (N=1 remontée gauche, N=2 remontée droite)

```
procedure parcours_iteratif(A : Arbre);
var P : Pile ; N : integer;
debut
        N ← 1; P ← pile-vide ;
        repeter
                si N=1 alors debut
                        tant que A <> ∅ faire debut
                                TRAITEMENT1
                                P← empiler(P, (A,1))
                                A ← g(A)
                        fin
                        TERMINAISON
                fin
                si non est-vide(P) alors debut
                        (A,N) ← sommet(P)
                        P ← dépiler(P)
                        si N=1  alors debut
                                TRAITEMENT2;
                                P← empiler(P, (A,2));
                                A ← d(A);
                        fin
                        sinon TRAITEMENT3;
                fin
        jusqu'a est-vide(P)
fin
```

5. Graphes

5.1 Graphes orientés

Un **graphe orienté** est constitué d'un ensemble N de **sommets** et d'une relation binaire A sur N (on confondra A et son graphe, c'est-à-dire l'ensemble des couples $(u,v) \in N \times N$ tels que u est en relation avec v). Un graphe G sera donc noté G = (N,A). L'ensemble A est appelé ensemble des **arcs** du graphe orienté. Un arc est donc une paire de sommets.

Si (u,v) est un arc, on dit que u est le **prédécesseur** de v et v est le **successeur** de u. On appelle u **l'extrémité initiale** de l'arc et v son **extrémité finale** (ou terminale)**.**

Comme pour les arbres, il est possible d'attacher une **étiquette** à chaque sommet du graphe. Il ne faut pas confondre nom d'un sommet et étiquette. Le nom d'un sommet doit être unique dans un graphe, mais 2 ou 3 sommets peuvent avoir la même étiquette. On peut aussi attacher une étiquette aux arcs du graphe.

Un **chemin** dans un graphe orienté est une liste de sommets $(v_1,v_2,...,v_k)$ tels qu'il existe un arc de chaque sommet vers le suivant. La longeur du chemin est k–1, le nombre d'arcs constituant le chemin.

Un **cycle** dans un graphe orienté est un chemin de longueur non nulle qui part et aboutit au même sommet. La longueur du cycle est la longueur du chemin.

On appelle **cycle élémentaire** (resp. chemin élémentaire) un cycle (resp.chemin) pour lequel aucun sommet n'apparaît plus d'une fois dans le cycle (resp.chemin), l'unique répétition est celle du sommet final.

On appelle **chemin simple** un chemin pour lequel aucun arc n'apparaît plus d'une fois dans le chemin.

On appelle **chemin eulérien** tout chemin simple contenant tous les arcs.

On appelle **chemin hamiltonien** tout chemin élémentaire contenant tous les sommets.

Etant donné un cycle non-élémentaire contenant le sommet v, on peut toujours trouver un cycle élémentaire contenant v.

Si un graphe a un ou plusieurs cycles, on dit qu'il est **cyclique**, dans le cas contraire on dit qu'il est **acyclique**.

Un chemin est dit **acyclique** (ou élémentaire) si aucun sommet n'apparaît plus d'une fois dans le chemin.S'il existe un chemin quelconque de u à v, alors il existe un chemin acyclique de u à v.

5.2 Graphes non-orientés

Quand la liaison entre deux sommets n'a pas de direction, on l'appelle une **arête**. Une arête est donc un ensemble de 2 sommets. Si {u,v} est une arête, on dit que u et v sont **adjacents**, ou bien **voisins.** Un graphe ayant des arêtes, c'est-à-dire tel que la relation des arcs est symétrique est appelé un **graphe non-orienté.**

On définit les **chemins** (ou chaînes) de façon analogue aux graphes orientés avec la différence que les sommets sont reliés par des arêtes. Trouver les **cycles** (ou circuits) dans des graphes non-orientés est un peu délicat car on ne veut pas appeler cycle un chemin (u,v,u) s'il existe une arête {u,v}. De même si $(v_1,v_2,...,v_k)$ est un chemin, on peut le suivre dans un sens ou dans l'autre, mais on ne veut pas appeler cycle le chemin $(v_1,v_2,...,v_{k-1},v_k,v_{k-1},...,v_2,v_1)$. On ne s'intéressera qu'à la

notion de **cycle élémentaire** dans un graphe non-orienté, qui est définit comme un chemin de longueur trois ou plus, qui commence et finit au même sommet, et ne répète aucun sommet à l'exception du premier.

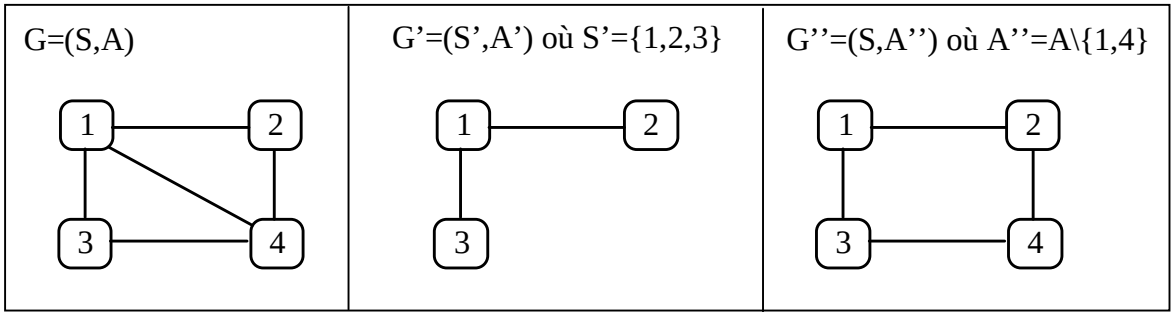
5.3 Definitions complémentaires

Un graphe **valué** orienté (resp. non-orienté) est un triplet (S,A,C) où S est un ensemble fini de sommets et où A est un ensemble fini d'arcs (resp. d'arêtes) et C une fonction de A dans $\mathbb{R}$ appelée coût.

Soit G = (S,A) un graphe. Le **sous-graphe de G engendré par S'⊆S** est le graphe G'= (S',A') dont les arcs (resp. arêtes) sont les arcs (resp. arêtes) de G ayant leurs 2 extrémités dans S'.

Soit G = (S,A) un graphe. Le **graphe partiel de G engendré par A''⊆A** est le graphe G''= (S,A'') dont les sommets sont les éléments de S et dont les arcs (arêtes) sont ceux de A''.

Exemple :



Un graphe non orienté (resp. orienté) est dit **complet** si pour tout couple de sommets (x,y), il existe une arête {x,y} (resp. un arc x→y).

Dans un graphe orienté, le nombre d'arcs sortants d'un sommet x est noté $d^{\circ +}(x)$ et s'appelle le **demi-degré extérieur de x.** On définit de façon similaire le **demi-degré intérieur** d'un sommet.

Dans un graphe orienté (resp.non-orienté) on appelle **degré** d'un sommet x et on note d°(x), le nombre d'arcs (resp.d'arêtes) dont x est une extrémité.

Un graphe orienté est dit **fortement connexe** si pour toute paire ordonnée de sommets distincts x et y, il existe un chemin de x vers y et un chemin de y vers x.

Un graphe non orienté est **connexe** si pour toute paire de sommets x ,y il existe une chaîne (un chemin constitué d'arêtes) reliant x et y.

On appelle **composante fortement connexe** d'un graphe orienté un sous-graphe fortement connexe maximal, c'est-à-dire un sous-graphe fortement connexe qui n'est pas strictement contenu dans un autre sous-graphe fortement connexe.

Dans un graphe non orienté, on appelle **composante connexe** un sous-graphe connexe maximal.

5.4 Structures de données

Il existe deux façons classiques de représenter les graphes :

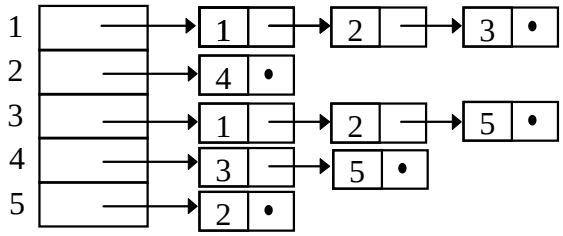
Listes d'adjacence :

```
type   list = ^cellule;
       cellule = record
```

```
                    nom     : type_som;
                    suivant : list
            end;
    graphe = array[1..nbsom] of list;
```

avec type_som le type de l'étiquette de chaque sommet, et nbsom le nombre de sommets.

Dessinez le graphe orienté représenté par les listes d'adjacence données ci-dessous :



Le demi-degré extérieur d'un sommet est égal à la longueur de sa liste d'adjacence.

Matrices d'adjacence : on représenter un graphe est par une matrice Adj de booléens. S'il existe un arc (une arête) menant du nœud i au nœud j alors Adj[i,j] est vrai, faux sinon. La matrice d'adjacence d'un graphe non-orienté est symétrique.
Voici la matrice d'adjacence correspondant aux listes d'adjacence données précédemment.

	1	2	3	4	5
1	t	t	t	f	f
2	f	f	f	t	f
3	t	t	f	f	t
4	f	f	t	f	t
5	f	v	f	f	f

Le demi-degré extérieur d'un sommet i est le nombre de "t" dans la colonne i de la matrice.

Comparaison des deux représentations :
La représentation d'un graphe de n sommets par sa matrice d'adjacence occupe un espace mémoire en n^2. Quand le graphe est peu dense (peu d'arêtes ou d'arcs), il faut lui préférer la représentation par les listes d'adjacence. Si on cherche à connaître la liste des successeurs d'un sommets, les deux représentations sont équivalentes. Par contre, si on cherche à savoir si deux sommets sont connectés, la représentation matricielle est bien plus adaptée.

5.5 Algorithmes élémentaires sur les graphes

Parcours en profondeur (depth-first search)
Partant d'un sommet, on suit un chemin le plus loin possible, puis on retourne en arrière pour reprendre tous les chemins ignorés précédemment.

```
{programme principal }
var i:integer ; Gr : graphe ; marque : array[1..n] of booelan ;
...
begin
        for i := 1 to n do marque[i] := false;
        for i := 1 to n do if not ( marque[i] ) then prof( i,gr,marque )
```

```
end;
{procédure récursive}
procedure prof( s:integer ; G:graphe ; var M : array [1..n] of boolean);
var j,v : integer ;
begin
        M[s] := true;
        for  j := 1 to  d o+-de( s,g ) begin
            v := eme-succ ( j,s,G );
            if not ( M[v] ) then prof( v,G,M )
        end;
end;
```

Parcours en largeur (breadth-first search)

Cet algorithme tient son nom au fait qu'il découvre d'abord tous les sommets situés à une distance k de s avant de découvrir tout sommet situé à la distance k+1. On utilise une structure de file : lorsqu'à partir d'un sommet s, on visite ses successeurs non marqués, il est nécessaire de les ranger successivement dans une file (FIFO) puisque la recherche au niveau suivant repartira de chacun des successeurs de s, à partir du premier.

```
procedure largeur(s : integer; G : graphe; var M : array[1..n] of boolean);
{s est un sommet}
var v,w,i : integer; {v et w sont des sommets}
begin
    F := file-vide ; M[s] := true ;
    F := ajouter( F,s ) ;
    while not ( est-vide( F )) do begin
        v := premier ( F ) ; F := retirer ( F );
        for  i := 1 to d o+-de( v, G ) do begin
            w := eme-succ ( i,v,G );
            if not ( M[w] ) then begin
                M[w] := true ; F := ajouter ( F,w );
            end;
        end;
    end;
end;
```

6. ALGORITHMES DIVISER POUR REGNER

6.1 Principe général

Un moyen classique d'aborder un problème est d'essayer de le découper en sous-problèmes, de résoudre les sous-problèmes pour ensuite combiner leurs solutions en une solution pour le problème global. Cette technique de résolution de problèmes est appelée **diviser pour régner** (*divide and conquer* que nous noterons DQ). Si les sous-problèmes sont similaires à l'original, alors on utilise la même procédure pour résoudre récursivement les sous-problèmes.
Cette technique fonctionne sous deux conditions : les sous-problèmes doivent être plus simples que le problème d'origine, après un nombre fini de subdivisions, on doit tomber sur un sous-problème que l'on sait résoudre complètement.

Exemple : conception de circuits logiques
On s'intéresse à la façon de réaliser un circuit composé de portes OU à deux entrées, qui teste si un registre de 32 bits a tous ses bits à 0. Ce test est réalisé par une porte OU à 32 entrées, une sortie 0 signifie que le registre contient 0, une sortie 1 signifie que le registre ne contient pas 0. Cependant, on suppose que l'on ne dispose que de portes OU à deux entrées. On a besoin de n–1 portes à deux entrées pour calculer le OU de n entrées. Une première f çon de réaliser un OU sur 32 entrées est la suivante :

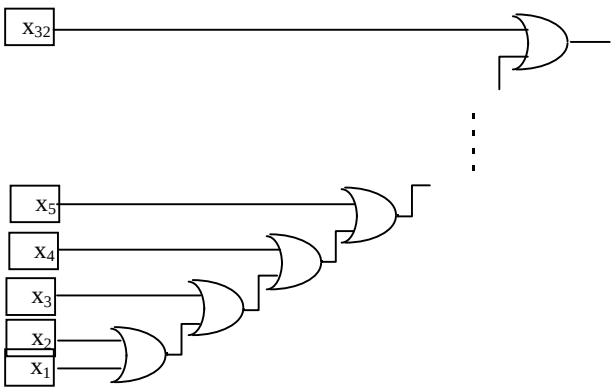


Figure 1 : façon inefficace de réaliser un OU sur 32 bits

Comme chaque porte alimente ma suivante et que l'on a 31 portes, le délai de ce circuit est 31. Une meilleure façon utilisant toujours 31 portes est illustrée par la figure 2. On peut remarquer que ce circuit est un arbre binaire complet. Son délai est de 5. On a obtenu ce circuit en utilisant le paradigme "diviser-pour-régner", c'est-à-dire, pour réaliser un OU de 2^k entrées, on divise les entrées en deux groupes de 2^{k-1} entrées chacun. Les circuits de chaque groupe sont combinés par une porte OU. Le circuit pour le cas de base k=1 (2 entrées) est réalisé en utilisant une porte OU à 2 entrées.

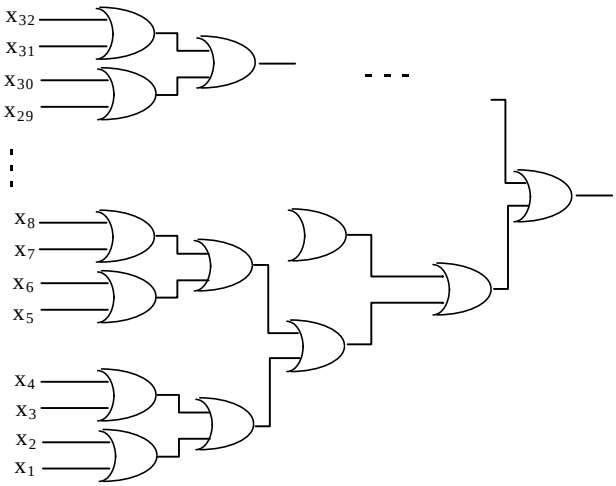


Figure 2 : approche diviser-pour-régner pour concevoir des circuits

Voici le schéma général d'un algorithme *diviser-pour-régner* :

```
fonction DQ(x)
  si taille(x)≤ seuil alors algo_base(x)
                      sinon décomposer(x,x1,x2,...,xk)
                            pour i←1 à k faire yi←DR(xi)
                            recombiner(y1,y2,...,yk,y)
                            DQ ←y
```

Si k = 1, on parlera plutôt de simplification au lieu de diviser-pour-régner. C'est le cas de l'algorithme récursif de calcul de factorielle. Quand k = 2, on parle souvent de dichotomie.

6.2 Analyse de diviser-pour-régner

Supposons que l'on ait un algorithme quadratique. Soit c une constante telle que l'implémentation de A ait une complexité en temps $t_A(n) \leq cn^2$ sur une instance de taille n. Supposons qu'il soit possible de résoudre cette instance en la décomposant en trois sous-instances de taille E(n/2), en résolvant ces trois sous-instances puis en combinant les solutions.
Soit d une constante telle que le temps pour décomposer et combiner vérifie $t(n) \leq d.n$.
En utilisant A et la décomposition on obtient un algorithme B dont l'implémentation aura une complexité

$$t_B(n) = 3\, t_A(E(n/2)) + t(n) = 3c(E(n/2))^2 + dn \leq 3c\,\frac{(n+1)^2}{2} + dn \leq \frac{3}{4}cn^2 + \left(\frac{3}{2}c + d\right)n + \frac{3}{4}c$$

sur de grandes instances (c'est le terme $\frac{3}{4}cn^2$ qui domine), B est 25% meilleur que A, mais reste quadratique.

Pour faire mieux, on va décomposer récursivement les sous-instances à leur tour jusqu'à une certaine taille n_0 .

La complexité de l'algorithme C ainsi obtenu est donnée par :

$$t_C(n) = \begin{cases} t_A(n) & \text{si } n \leq n_0 \\ 3\, t_C(E(n/2)) + t(n) & \text{sinon} \end{cases}$$

Le choix du seuil (taille des sous-instances pour laquelle on arrête de décomposer) est un problème délicat que nous n'aborderons pas. Il faut quand même savoir que si le seuil n'intervient pas dans l'ordre de grandeur de la complexité de l'algorithme de diviser-pour-régner, il ne faut pas le négliger pour autant comme le prouve l'exemple suivant.

Exemple : pour déterminer le seuil n_0 qui minimise $t_C(n)$, il ne suffit pas de savoir que $t_A(n) \in \Theta(n^2)$ et que $t(n) \in \Theta(n)$. Considérons une implémentation telle que $t_A(n) = n^2$ millisecondes et $t(n) = 16n$ millisecondes. Supposons que l'on ait à résoudre une instance de taille 1024. Si l'algorithme procède récursivement jusqu'à obtenir des instances de taille 1 ($n_0 = 1$), il mettra plus d'une demi-heure pour résoudre le problème, ce qui est ridicule car l'algorithme de base A (en prenant $n_0 =$ '∞') résoud le problème en un peu plus d'un quart d'heure ! En fait, si l'on prend un 'bon' seuil, (n_0=64), l'algorithme C résoud l'instance de taille 1024 en moins de 8 minutes.

6.3 D-p-R appliqué au tri

6.3.1 Tri fusion

Soit L un tableau de n éléments que l'on souhaite trier en ordre croissant. Une façon évidente d'utiliser la stratégie diviser-pour-régner consiste à diviser le tableau en 2 parties égales (à un élément près si n est impair), trier ces deux parties par des appels récursifs, puis fusionner les solutions pour les deux parties, en faisant attention de préserver l'ordre des éléments. Voici l'algorithme :

```
procedure tri_fusionl(L[1...n])
  si n est petit alors insertion(L)
  sinon (* U et V tableaux de taille n div 2, (n+1)div 2 resp.*)
        recopie( U, L[1 ... n div 2] )
        recopie( V, L[1+(n div 2) ... n])
        tri_fusion(U)
        tri_fusion(V)
        fusionner(L,U,V)
```

où fusionner est une procédure qui intercale deux sous-séquences triées de tailles n et m.

Exercice : écrire la procédure fusionner et déterminer sa complexité. Analyser ensuite l'algorithme de tri-fusion.

6.3.2 Tri rapide (quick sort)

Le tri rapide est également basé sur une stratégie diviser-pour-régner. La partie non récursive de l'algorithme est ici consacrée à la division (pour le tri fusion, c'est à la phase de combinaison qu'est consacrée cette partie non-récursive).

```
procedure tri_rapide (A[1...n], i, j)
      si  i < j alors   partionnement ( A [ i..j ], k )
                        tri_rapide ( A,  i, k-1 )
                        tri_rapide ( A, k+1, j )
```

```
procedure partionnement (A [ i..j ], k )
        p := A [ i ]
        q := i
        k := j + 1
        répéter q := q +1 jusqu'à ( A[ q ] > p ou q >= j )
répéter k := k - 1 jusqu'à ( A [ k ] <= p )
        tant que  q < k  faire
                échange(A [ q ], A [ k ])
                répéter q := q + 1 jusqu'à A [ q ] > p
                répéter k :=  k - 1 jusqu'à A [ k ] <= p
        échange(A [ i ], A [ k ])
```

3 étapes du processus « diviser pour régner » pour trier un sous-tableau A[i..j] :

- **diviser :** A[i..j] est partionné en 2 sous-tableaux non vides A[i..k] et A[k+1..j] tels que chaque élément de A[i..k] soit inférieur ou égal à chaque élément de A[k+1..j]. L'indice k est calculé pendant la procédure de partionnement.
- **régner :** les 2 sous-tableaux A[i..k] et A[k+1..j] sont triés par des appels récursifs à la procédure principale du tri rapide.
- **combiner :** comme les sous tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner : A[i..j] tout entier est trié

Exemple: appel initial tri_rapide(A,1,longueur(A))

6	2	8	5	10	9	12	1	15	7	3	13	4
6	2	4	5	10	9	12	1	15	7	3	13	8
6	2	4	5	3	9	12	1	15	7	10	13	8
6	2	4	5	3	1	12	9	15	7	10	13	8
1	2	4	5	3	6	12	9	15	7	10	13	8

Le temps d'exécution de cet algorithme dépend de l'instance et du choix du pivot.
Que vaut k dans partionnement quand tous les éléments du tableau sont égaux?
Quel est le cas le pire ? Quel est le meilleur des cas ?

Calcul de la complexité dans le pire des cas :
Le pire des cas intervient quand partionnement produit une région avec n-1 éléments et une autre avec un seul élément. Le partionnement coûte $\Theta(n)$ la récurrence pour le temps d'exécution est donc : $T(n) = T(n-1) + \Theta(n)$.

Evaluer cette récurrence, en observant que $T(1) = \Theta(1)$.

Calcul de la complexité dans le meilleur des cas :
Si la procédure partionnement produit deux régions de taille n/2, le tri rapide s'exécute beaucoup plus rapidement (!). La récurrence s'écrit alors : $T(n) = 2T(n/2) + \Theta(n)$.

Evaluer cette récurrence.

Complexité en moyenne : on peut montrer que le tri rapide a un coût en moyenne de l'ordre de O(nlogn).

6.4 Autres exemples

6.4.1 Recherche dichotomique dans une liste triée

La recherche dichotomique est l'algorithme que l'on utilise pour chercher un mot dans un dictionnaire, et c'est une des applications les plus simples du principe diviser-pour-régner. On s'intéresse au problème de rechercher une valeur x dans un tableau T (ou une liste) de n valeurs triées par ordre croissant. Si x n'est pas dans le tableau, on voudrait la position où il faudrait que x soit insérée. On veut donc l'indice i tel que $1 \leq i \leq n$ et $T[i] \leq x < T[i+1]$.

On a déjà vu un algorithme de recherche séquentielle, qui consiste à parcourir le tableau du plus petit élément vers le plus grand jusqu'à trouver x ou un élément supérieur à x. Cet algorithme a une complexité en $\theta(1+i)$ (où i est l'indice déterminé par la recherche), $\Omega(n)$ dans le pire des cas et $O(1)$ dans le meilleur des cas.

Le principe D-p-R suggère de vérifier d'abord si x est dans la première moitié du tableau ou dans la seconde. Pour le savoir, il suffit de comparer x à la valeur qui se trouve au milieu du tableau (T[n div 2]). Voici l'algorithme, dans sa version récursive :

```
procédure rech_dicho(X : Element; T : array [1..n] of Element; g,d : 0..n+1);
var m : 1..n;
begin
  if g ≤ d then begin
        m ← (g+d) div 2;
        if X = T[m] then res ← m
                  else if X < T[m] then rech_dicho(X, T, g ,m-1, res)
                                   else rech_dicho(X, T, m+1, d, res);
        end
        else res := g ;
end;
```

Cet algorithme est en $\theta(\log n)$. Il faut noter qu'un seul des deux appels récursifs est exécuté, c'est donc un exemple de simplification plutôt que de diviser-pour-régner.

6.4.2 Multiplication de matrices (algorithme de Strassen)

Rappelez l'algorithme classique de la multiplication de matrices et donnez le nombre d'opérations élémentaires (additions/multiplications scalaires).
Nous supposons que $n=2^k$. On va diviser A, B et C chacune en quatre sous-matrices n/2 x n/2 :

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

où la matrice C_{ij} (n/2 x n/2) est définie par : $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$, $1 \leq i, j \leq 2$.

Pour n = 2, les sous matrices sont de taille 1x1, le produit AB est direct. Pour n > 2, la multiplication des sous-matrices se fait par application récursive de la division des matrices en quatre sous-matrices.

Donnez les étapes d'un algorithme récursif qui implémente le principe DQ décrit ci-dessus pour la multiplication de matrices. Ecrivez et résolvez la formule récurrence pour le temps de cet algorithme.

Strassen a découvert une méthode qui utilise 7 multiplications récursives de matrices n/2 x n/2 (au lieu de 8). Prenons le cas de matrices 2x2 :

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

posons :
$m_1 = (a_{21} + a_{22} - a_{11})(b_{22}- b_{12} + b_{11})$
$m_2 = a_{11}\, b_{11}$
$m_3 = a_{12}\, b_{21}$
$m_4 = (a_{11} - a_{21})(b_{22}- b_{12})$
$m_5 = (a_{21} + a_{22})(b_{12}- b_{11})$
$m_6 = (a_{12} - a_{21} + a_{11}- a_{22})\, b_{22}$
$m_7 = a_{22}(b_{11}+ b_{22} - b_{12} - b_{21})$.

Vérifier que la matrice C = AB s'écrit en fonction des m_i (i= 1, ... , 7) de la façon suivante :

$$\begin{bmatrix} m_2 + m_3 & m_1+m_2+m_5+m_6 \\ m_1+m_2+m_4-m_7 & m_1+m_2+m_4+m_5 \end{bmatrix}$$

On a donc multiplié deux matrices 2x2 en utilisant 7 multiplications scalaires et 14 additions ou soustractions (4 pour l'algorithme classique). A priori, il semble que l'on n'ait rien gagné.
Pour généraliser au cas de matrices A et B nxn, on remplace chaque terme de A et B par une matrice n/2 x n/2. L'algorithme obtenu multiplie deux matrices nxn en faisant 7 multiplications de matrices n/2 x n/2 (et un certain nombre d'additions).
Montrez que l'algorithme de Strassen multiplie deux matrices nxn en un temps en $\theta(n^{\ln 7}) = \theta(n^{2,81})$.

7. PROGRAMMATION DYNAMIQUE

7.1 Problèmes d'optimisation

Dans ce type de problèmes, il peut y avoir de nombreuses solutions. A chaque solution est affectée une valeur, et on souhaite trouver la solution ayant la valeur optimale (minimum ou maximum). Une telle solution est **une** solution optimale.

Exemples : trouver le meilleur ordre pour exécuter un ensemble de tâches, trouver le plus court chemin dans un graphe, problème du sac-à-dos, ...

En général, on a :
- un ensemble de candidats,
- un ensemble de candidats déjà choisis,
- une fonction qui indique si un ensemble de candidats constitue une solution, sans considération d'optimalité,
- une fonction qui détermine si un ensemble de candidats est **faisable**, c'est-à-dire s'il est possible de le compléter de manière à obtenir au moins une solution (pas nécessairement optimale),
- une fonction de sélection qui indique à chaque instant le candidat le plus prometteur, parmi les candidats qui n'ont pas encore été choisis,
- une fonction **objectif** qui donne la valeur d'une solution. C'est cette fonction que l'on cherche à optimiser.

La programmation dynamique est une méthode qui s'applique aux problèmes d'optimisation respectant la propriété de sous-structure optimale : dans une séquence de choix optimale, chaque sous-séquence doit être optimale.
C'est une technique *bottom-up* qui résout des sous-instances minimales, les combine, obtenant ainsi la solution de sous-instances de taille croissante, pour finalement arriver à la solution de l'instance globale. Nous avons vu des techniques **diviser pour régner** qui sont des méthodes *top-down*.

Pour certains problèmes, la division en sous-problèmes conduit naturellement à considérer plusieurs sous-problèmes qui se « chevauchent ». Si l'on résout chacun d'entre eux indépendamment, les mêmes calculs seront faits plusieurs fois. Si, au contraire, on profite de ces recouvrements (en conservant les résultats intermédiaires), il est probable que l'on aura un algorithme plus efficace.
C'est l'idée de base de la programmation dynamique : éviter de calculer plusieurs fois la même chose, en utilisant une table des résultats connus, actualisée au fur et à mesure de la résolution des sous-problèmes.

Exemple 1 : Calcul des coefficients binômiaux

```
fonction C(n,k)
    si k = 0 ou k = n alors C= 1
    sinon C = C(n-1, k-1) + C(n-1, k)
```

Le nombre total d'appels récursifs pour le calcul de C(n,k) est $2C_n^k - 2$.

En utilisant une table contenant les résultats intermédiaires, on obtient un algorithme bien plus efficace. Notez qu'il suffit de stocker la ligne courante, que l'on remplit de droite à gauche.

Exemple 2 : Propriété de sous-structure optimale (ou principe d'optimalité)
le problème de trouver le plus court chemin entre 2 points d'un graphe a la propriété de sous-structure optimale, ce n'est pas le cas du problème qui consiste à trouver le plus long chemin entre 2 points.

7.2 Problème du plus court chemin et algorithme de Floyd

Soit G = <N, A> un graphe orienté. Un coût est associé à chaque arc, et l'on veut calculer le coût associé au plus court chemin entre chaque paire de sommet.

On définit L la matrice des coûts par : $L[i, i] = 0$
$L[i, j] \geq 0$ si $i \neq j$ et $(i, j) \in A$
$L[i, j] = \infty$ si $(i, j) \notin A$

Nous construisons la matrice D du plus court chemin entre chaque paire de sommets ($D = D_n$) :
D_0 est initialisée à L,
à la kième itération, D_k donne la longueur des plus courts chemins passant par les sommets {1,2, ..., k} : $D_k [i, j] = \min(D_{k-1} [i, j], D_{k-1} [i, k] + D_{k-1} [k, j])$.

Une 2ème matrice P permet de savoir par quels sommets passent les chemins. P est initialisée à la matrice nulle, et chaque fois qu'on ajoute un sommet à un chemin, on actualise P.

```
procédure Floyd (L, D, P)
  D ← L
  pour k ← 1 à n faire
        pour i ← 1 à n faire
              pour j ← 1 à n faire
                    si D[i, j] > D[i, k] + D[k, j] alors
                          D[i, j] ← D[i, k] + D[k, j]
                          P[i, j] ← k
```

7.3 Multiplication de matrices (parenthésage)

Rappelez combien de multiplications élémentaires sont nécessaires pour le calcul du produit si A matrice (p,q) et B matrice (q,r) ?
Supposons que l'on veuille multiplier les matrices $M_1(30,35)$, $M_2(35,15)$, $M_3(15,5)$ et $M_4(5,10)$. Rappelons que la multiplication des matrices est associative. Combien de parenthésages possibles pour le calcul de $M = M_1M_2M_3M_4$? Donnez pour chaque parenthésage, le nombre de multiplications élémentaires effectuées.

Nous nous intéressons maintenant au problème suivant : étant donnée une suite de n matrices : $M = M_1M_2 \ldots M_n$, où pour tout i, la matrice M_i a une dimension (d_{i-1}, d_i), parenthéser complètement le produit $M_1M_2 \ldots M_n$ de façon à minimiser le nombre de multiplications scalaires. Pour trouver la meilleure façon de multiplier n matrices, on pourrait parenthéser l'expression de toutes les façons possibles et évaluer le coût pour chaque parenthésage. Soit P(n) le nombre de parenthésages différents pour le produit de n matrices, on a :

$$P(n) = \sum_{i=1}^{n-1} P(i)\, P(n-i) .$$

On peut montrer que $P(n) = \frac{1}{n} C_{2n-2}^{n-1} \geq \frac{4^{n-1}}{2n}$.

Comment s'exprime ici la propriété de sous-structure optimale ?

Nous allons construire un tableau C de dimension (n,n) tel que C[i,j] donne la solution optimale pour le sous produit $M_i M_{i+1} ... M_j$. Le coût de la solution optimale sera donc donné par C[1,n].
On construit la matrice C diagonale par diagonale : la diagonale s contient les élément C[i, j] tels que j-i = s.

(s = 0)	C[i, i], i=12,.., n
(s = 1)	C[i, i+1], i=1,2,..., n-1
(1 < s < n)	C[i, i+s], i=1,2,..., n-s

On rappelle que M_i est de dimension (d_{i-1}, d_i). Ecrivez l'algorithme pour calculer C[1,n]. Analyser votre algorithme.
Comment pourrait-t-on modifier l'algorithme de calcul de C[1,n] pour qu'en plus il donne le parenthésage optimal ?

Exemple :

matrice	**dimension**
M_1	30 x 35
M_2	35 x 15
M_3	15 x 5
M_4	5 x 10
M_5	10 x 20
M_6	20x 25

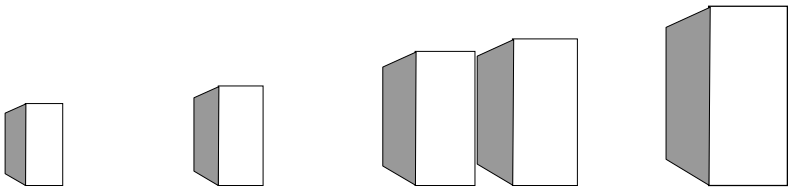
parenthésage		coût
((M1M2)M3)M4)	(30*35*15)+(30*5*10)+(30*15*5)	19500
(M1M2)(M3M4)	(30*35*15)+(15*5*10)+(30*15*10)	21000
(M1(M2M3))M4	(35*15*5)+(30*35*5)+(30*5*10)	9375
M1((M2M3)M4)	(35*15*5)+(35*5*10)+(30*35*10)	14875
M1(M2(M3M4))	(15*5*10)+(35*15*10)+(30*35*10)	16500

Tableau C

	1	**2**	**3**	**4**	**5**	**6**
1	0	15 750	7 875	9 375	11 875	15 125
2		0	2 625	4 375	7 125	10 500
3			0	750	2 500	5 375
4				0	1 000	3 500
5					0	5 000
6						0

7.4 Problème du sac-à-dos

Un voleur dévalise un coffre-fort qui contient N types d'objets de différentes tailles et de différentes valeurs. Il n'a qu'un sac-à-dos de contenance M pour emporter son butin. Le fameux **problème du sac-à-dos** consiste à trouver l'ensemble d'objets que le voleur devrait prendre dans son sac de façon à maximiser la valeur totale de son butin.
Par exemple, supposons que la capacité du sac soit 17 et que le coffre-fort contienne beaucoup d'objets de 5 types décrits ci-dessous :



taille	**3**	**4**	**7**	**8**	**9**
valeur	**4**	**5**	**10**	**11**	**13**
nom	**A**	**B**	**C**	**D**	**E**

Avec ces données, le voleur peut prendre 5 objets de type A (mais pas 6) pour une valeur de 20, ou bien un D et un E pour un total de 24. Il a encore beaucoup d'autres possibilités, mais il voudrait bien connaître celle qui lui permettra de réaliser la meilleure valeur.

Voici un algorithme de programmation linéaire qui résout ce problème :

```
pour j ← 1 à N faire
début
        pour i ← 1 à M faire
        début
            si  i – taille[j] ≥ 0 alors
                si coût[i] < ( coût[i–taille[j]] + val[j]) alors
                        coût[i] ← coût[i–taille[j]] + val[j]
                        meilleur[i] ← j
        fin
fin
```

8. La stratégie gloutonne

Les algorithmes gloutons n'aboutissent pas toujours à une solution optimale, mais y arrivent dans la plupart des cas. Ils suivent une stratégie heuristique :

- l'ensemble des candidats choisis est vide au départ,
- à chaque étape, on essaie d'ajouter à cet ensemble le meilleur des candidats restants, ce choix étant guidé par la fonction de sélection,
- si l'ensemble ainsi constitué, n'est pas faisable, on enlève le dernier candidat ajouté (et on ne le considérera plus),
- si l'ensemble est faisable, le candidat ajouté y restera jusqu'à la fin (on ne remet jamais en question un choix, pas de retour arrière).

Propriété du choix glouton :
Un problème a la propriété dite du choix glouton si l'on peut arriver à une solution globalement optimale en effectuant un choix localement optimal.

Sous-structure optimale :
Un problème fait apparaître une sous-structure optimale si une solution du problème contient la solution optimale de sous-problèmes.

8.1 Problème du choix d'activités

Il s'agit de répartir une ressource parmi plusieurs activités en compétition. Pour ce problème, un algorithme glouton fournit une méthode élégante pour trouver un ensemble le plus grand possible d'activités compatibles entre elles.

- S = {1,2, ..., n}, ensemble de n activités concurrentes, utilisant une ressource commune ; chaque activité i a un horaire de début d_i et de fin f_i , avec $d_i \leq f_i$.
- Les activités i et j sont compatibles si les intervalles $[d_i , f_i [$ et $[d_j , f_j [$ sont disjoints.
- Le problème du choix d'activités consiste à trouver **le plus grand nombre** possible d'activités compatibles entre elles.

On suppose que les activités sont triées par horaires de fin croissants : $f_1 \leq f_2 \ldots \leq f_n$

```
algorithme choix-d'activités-glouton
    n ← longueur (S)
    A ← {1}
    j ← 1
    pour i ← 1 à n faire
            si d_i ≥ f_j  alors   A ← A ∪ {i}
                                  j ← i
```

Cet algorithme est très efficace puisqu'il peut ordonnancer un ensemble de n activités en $\theta(n)$, si les activités sont triées selon leur horaire de fin.
Le choix glouton est celui qui maximise la quantité de temps libre restant (l'activité choisie est celle ayant l'horaire de fin le moins tardif).

Validité de l'algorithme glouton pour le problème du choix d'activités :

L'activité 1 a l'horaire de fin le moins tardif. On souhaite montrer qu'il existe une solution optimale qui commence par un choix glouton, c'est-à-dire par l'activité 1.
Supposons que $A \subseteq S$ est une solution optimale pour l'instance donnée et que la première activité de A soit l'activité k.

Si k=1, alors l'ordonnancement A commence par un choix glouton.
Si $k \neq 1$, on montre qu'il existe une autre solution optimale $B = A - \{k\} \cup \{1\}$. Comme $f_1 \leq f_k$ les activités de B sont disjointes, et comme B a le même nombre d'éléments que A, B est optimale.
On a donc montré qu'il existait toujours un ordonnancement optimal commençant par un choix glouton.

Une fois que le choix de l'activité 1 est fait, le problème se ramène à trouver une solution optimale pour le problème du choix d'activités de l'ensemble S compatibles avec l'activité 1.
On a donc que si A est une solution optimale pour S, alors $A' = A - \{1\}$ est une solution optimale pour $S' = \{i \in S, d_i \leq f_1\}$. En effet si l'on pouvait trouver B' une solution optimale pour S' contenant plus d'activités que A', ajouter 1 à B' offrirait une solution optimale pour S contenant plus d'activités que A, ce qui contredirait l'hypothèse que A est optimale.

Donc après chaque choix glouton, on se retrouve avec un problème d'optimisation de la même forme que le problème initial. Par induction sur le nombre de choix effectués, faire le choix glouton à chaque étape produit une solution optimale.

8.2 Arbre couvrant minimal

Soit G = (V, A) un graphe non-orienté connexe, avec V ensemble des sommets et A ensemble des arêtes. A chaque arête est associé un coût ≥ 0 (distance). Le problème consiste à trouver un sous-ensemble T de A tel que tous les sommets de V restent connectés par les arêtes de T, et la somme des coûts des arêtes de T est minimale.

On peut prouver que le graphe (V, T) est un arbre. Cet arbre est appelé arbre couvrant minimal.

```
algorithme-de-Kruskal
  trier A en ordre croissant
  n ← # V
  T ← ∅
  C ← {{i}, i ∈ V}
  répéter
          < u, v > ← la plus petite arête de A
          si ( u et v sont dans des ensembles distincts de C, R1 et R2 ) alors
                  remplacer R1 et R2 par R1∪ R2 dans C
                  ajouter  < u, v > dans T
          enlever  < u, v > dans A
  jusqu'à ce que #  = n-1
```

L'algorithme de Kruskal est en O (a log a) avec a = # A

8.3 Plus courts chemins

Soit G = (V, A) un graphe orienté, avec V ensemble des sommets et A ensemble de arcs. A chaque arc est associé un coût ≥ 0. Le problème consiste à déterminer le coût des plus courts chemins reliant un sommet de départ à chacun des autres sommets de V.

On supposera que les sommets sont numérotés de 1 à n et que le sommet 1 est le sommet de départ.
La matrice L donne le coût associé à chaque arc : $L[i, j] \geq 0$ si $< i, j >$ est dans A
$L[i, j] = \infty$ si $< i, j >$ n'est pas dans A

```
algorithme-de-Dijkstra
        C ← {2,3, ..., n }
        E ← ∅
        pour i ← 2 à n faire D[ i] ← L[ 1, i]
        pour i ← 2 à n faire
                v ← élément de C avec une valeur D[ v] minimale
                C ← C - { v}
                E ← E ∪ {v}
                pour chaque w ∈ C faire
                        D[ w] ← min( D[ w], D[ v] + L[ v, w]
```

9. Introduction à la théorie de la complexité

9.1 Introduction et motivation

Pour motiver le contenu de cette partie du cours, on va s'intéresser à un problème de la théorie des graphes, le fameux problème du voyageur de commerce (en anglais, *Travelling Salesman Problem*). Etant donné un graphe non orienté, complet, valué à *n* sommets, il s'agit de trouver un cycle hamiltonien de longueur minimale. On rappelle qu'un cycle hamiltonien et un cycle qui passe une fois et une seule par chaque sommet, sa longueur est la somme des valuations des arêtes le constituant. Il s'agit donc d'un problème d'optimisation (dans un graphe complet, on sait qu'il existe $\frac{(n-1)!}{2}$ cycles hamiltoniens (sauriez-vous le prouver ?)

Un algorithme trivial pour résoudre ce problème consiste à énumérer toutes les solutions. On sait bien, vu le nombre de solutions faisables que ce n'est pas raisonnable (pour un graphe à 20 sommets, il faudrait 19 siècles de calcul à une machine capable de traiter 1 million de cycles à la seconde ![1]
Le problème, c'est qu'on ne connaît pas de meilleur algorithme, on peut accélérer la vitesse de résolution par des méthodes que nous ne verrons pas dans le cadre de ce cours, mais tous les algorithmes exacts (c'est-à-dire qui trouvent effectivement une solution optimale) sont impraticables.

Analyse d'algorithme : on sait prouver qu'un algorithme peut résoudre un certain problème en un temps en O(f(n)) pour une certaine fonction f.

Théorie de la complexité : on cherche à trouver une fonction g telle que **tout** algorithme résolvant le problème (pour toutes ses instances) sera nécessairement en $\Omega(g(n))$. On est satisfait quand $f(n) \in \Theta(g(n))$, car on sait alors qu'on a le meilleur algorithme possible. Dans ce cas on dit que la complexité du problème est connue exactement. Malheureusement il est rare que cela arrive !
Comme il est difficile de déterminer la complexité exacte d'un problème, on se contente souvent de comparer les difficultés relatives de différents problèmes :

- supposons qu'un certain nombre de problèmes sont équivalents dans le sens qu'ils ont la même complexité. Toute amélioration de la méthode de résolution d'un de ces problèmes mènera automatiquement (du moins en théorie) à un meilleur algorithme pour tous les autres.
- d'un point de vue négatif, si on connaît des problèmes pour lesquels aucun algorithme efficace n'a été trouvé, le fait que ces problèmes soient équivalents conforte l'idée qu'il n'existe probablement pas d'algorithme efficace pour ces problèmes.

Trois versions des problèmes d'optimisation :
1. Problème d'optimisation :trouver la solution optimale
2. Problème d'évaluation : trouver le coût de la solution optimale
3. Problème d'existence : déterminer s'il existe une solution optimale

Les trois versions sont considérées comme étant équivalentes, du point de vue de l'existence d'algorithmes efficaces. La théorie de la complexité traite de problèmes d'existence car leur

[1] $\frac{19!}{2} * 10^{-6} \approx 6 * 10^{10}$ sec ≈ 1902 années (31536000 secondes en un an)

formalisme à réponse O/N, V/F a permis de les étudier avec des outils de la logique mathématique.

Pour l'exemple du TSP : le problème d'optimisation consiste à donner le cycle hamiltonien le plus court, le problème d'évaluation consiste à donner la longueur du cycle hamiltonien le plus court, le problème d'existence consiste à répondre, étant donné un entier *k*, s'il existe un cycle hamiltonien de longueur inférieure à *k*.

9.2 Classification des problèmes d'existence

problèmes résolus par des algorithmes en O(n) ;

P problèmes résolus par des algorithmes en O(p(n)) où p(n) est un polynôme de degré supérieur à 1

NP problèmes pour lesquels on ne connaît pas d'algorithme polynomial, et qui apparemment n'ont pas de difficulté intrinsèque (c'est-à-dire que l'on peut vérifier en un temps polynomial qu'une solution proposée permet de répondre à la question posée)
problèmes qui requièrent, intrinsèquement, un nombre d'opérations égal à une fonction exponentielle en la taille des instances.

La classe **P** contient les problèmes que l'on peut résoudre en temps polynomial. Cette classe peut être définie de façon très précise avec le formalisme des machines de Turing. Des exemples de problèmes dans **P** :

étant donné *G* un graphe, *G* est-il connexe ?

étant donnés un graphe connexe valué et un entier *i*, existe-t-il un arbre de recouvrement de coût inférieur ou égal à *i* ?

Pour qu'un problème soit dans la classe NP, si *x* est une instance *oui* du problème, alors il existe une preuve « courte » que *x* est valide.

9.3 Transformations polynomiales

On dit qu'un problème d'existence A_1 est polynomialement transformable en un autre problème d'existence A_2 si, étant donnée une instance quelconque *x* de A_1, on peut construire une instance *y* en un temps polynomial (en taille(*x*)) telle que *x* soit une instance *oui* de A_1 si et seulement si *y* est une instance *oui* de A_2. On note $A_1 \propto_P A_2$.

Définition : Un problème d'existence A de la classe NP est dit NP-complet si tous les autres problèmes de NP sont polynomialement transformables en A, (pour le problème d'optimisation correspondant, on dit NP-difficile).

La classe des problèmes NP-complets a les deux propriétés suivantes :
1. aucun problème NP-complet ne peut être résolu par un algorithme polynomial connu
2. s'il existait un algorithme polynomial pour n'importe lequel des problèmes NP-complets, alors il existerait des algorithmes polynomiaux pour tous les problèmes de cette classe.

Conjecture : P ≠ NP

Un chercheur étudiant un nouveau problème et qui ne trouve pas de méthode de résolution efficace tentera de montrer que son problème est NP-complet.

Pour prouver qu'un problème est NP-complet il faut :
a) prouver qu'il est dans NP

b) prouver que tous les problèmes de NP sont polynomialement transformables en ce problème.
En pratique b) consiste à montrer qu'un problème NP-complet connu est polynomialement transformable en le problème qui nous intéresse (la relation *polynomialement transformable* est transitive).

9.4 Théorème de Cook (1971)

Un problème posé par la définition d'un problème NP-complet est le suivant : existe-t-il des problèmes NP-complets ? Le théorème de Cook donne une réponse positive à cette question.

Une expression booléenne est satisfiable s'il existe au moins une assignation de ses variables qui la rende *vraie*.
Une tautologie est une expression booléenne toujours *vraie*.
Une contradiction est une expression booléenne non satisfiable.

Exemples :	$(p \vee q) \Rightarrow p \wedge q$	est satisfiable
	$(p \Leftrightarrow q) \Leftrightarrow (\neg p \vee q) \wedge (p \vee \neg q)$	est une tautologie
	$\neg p \wedge (p \vee q) \wedge \neg q$	est une contradiction

SAT, TAUT, CONT sont les problèmes qui consistent à répondre si une expression donnée est, respectivement, satisfiable, une tautologie ou une contradiction.

Pour prouver qu'une expression booléenne est satisfiable, il suffit d'exhiber une assignation de ses variables qui la rende *vraie*. Pour cela on peut observer toutes les assignations possibles ce qui est impraticable quand le nombre de variables booléennes est grand (2^n assignations possibles). Par contre, étant données une assignation, il est immédiat de vérifier qu'elle rend l'expression *vraie* ou pas (donc SAT $\in$ NP).

Il est possible de prouver que SAT $\propto_P$ TAUT $\propto_P$ CONT

Un littéral est ou bien une variable booléenne ou bien sa négation
Une clause est un littéral ou une disjonction ($\vee$) de littéraux
Une expression booléenne est sous forme conjonctive normale (FCN) si c'est une clause ou une conjonction ($\wedge$) de clauses
Une expression booléenne est sous forme k-FCN ($k \in$ IN) si elle est composée de clauses, chacune avec un maximum de k littéraux

On peut prouver que toute expression booléenne peut être transformée en une expression FCN. SAT-FCN est la restriction du problème SAT aux expressions FCN. Pour tout entier *k*, SAT-*k*-FCN est la restriction de SAT-FCN aux expressions *k*-FCN. SAT-2-FCN peut être résolu en temps polynomial.

Théorème de Cook : SAT-FCN est NP-complet

9.5 Six exemples de problèmes NP-complets

3-SAT :
Instance : un ensemble $C = \{c1,c2,...,cm\}$ de clauses sur un ensemble fini U de variables booléennes telles que $|ci| = 3$ pour $1 \leq i \leq m$.
Question : existe-t-il une assignation des variables de U qui satisfasse toutes les clauses de C ?

3-DM : (Dimensional Matching)

Instance : un ensemble $M \subseteq W$ x X x Y, où W ,Y et Y sont des ensembles disjoints ayant le même cardinal q.
Question : l'ensemble M contient-il un matching (mariage), c'est-à-dire un sous-ensemble $M' \subseteq M$ tel que $|M'| = q$ et aucune paire d'éléments de M' n'a deux coordonnées égales ?

VC : (Vertex Cover ou couverture de sommets)
Instance : un graphe $G = (V, E)$ et un entier positif $K \leq |V|$.
Question : existe-t-il un recouvrement de taille inférieure ou égale à K, c'est-à-dire un sous-ensemble $V' \subseteq V$ tel que $|V'| \leq K$ et, pour toute arête $\{u,v\} \in E$, au moins l'un des deux sommets u ou v est dans V' ?

CLIQUE :
Instance : un graphe $G = (V, E)$ et un entier positif $J \leq |V|$.
Question : G contient-il une clique de taille supérieure ou égale à J, c'est-à-dire un sous-ensemble $V' \subseteq V$ tel que $|V'| \geq J$ et, toute paire de sommets de V' est reliée dans E ?

HC : (Circuit Hamiltonien)
Instance : un graphe $G = (V, E)$.
Question : G contient-il un circuit hamiltonien, c'est-à-dire une suite ordonnée $<v_1,v_2,...v_n>$ des sommets de G ($n=|V|$), telle que $\{v_n, v_1\} \in E$ et $\{v_i, v_{i+1}\} \in E$ pour tout $1 \leq i \leq n$?

PARTITION :
Instance : un ensemble fini A et une "taille" $s(a) \in Z^+$ pour tout $a \in A$.
Question : existe-t-il un sous-ensemble $A' \subseteq A$ tel que : $\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a)$?

9.6 Le problème de la clique est NP-complet

Théorème : **CLIQUE** est NP-complet

Preuve : CLIQUE $\in$ NP, en effet, étant donnée une instance du problème (un graphe G = (V,E) et k$\in$ IN) et C $\subseteq$ V, l'algorithme qui vérifie si C est une clique (tous les sommets de C connectés dans E) et si |C|=k est polynomial (en $\Theta(n^2)$).

Nous allons montrer que 3-SAT $\propto_P$ CLIQUE.

Une assignation partielle t assigne la valeur *vrai* ou *faux* a certaines variables seulement, les autres ont une valeur indéterminée t(x) = *d*. Une assignation partielle sera définie comme une suite de 0,1 ou d :

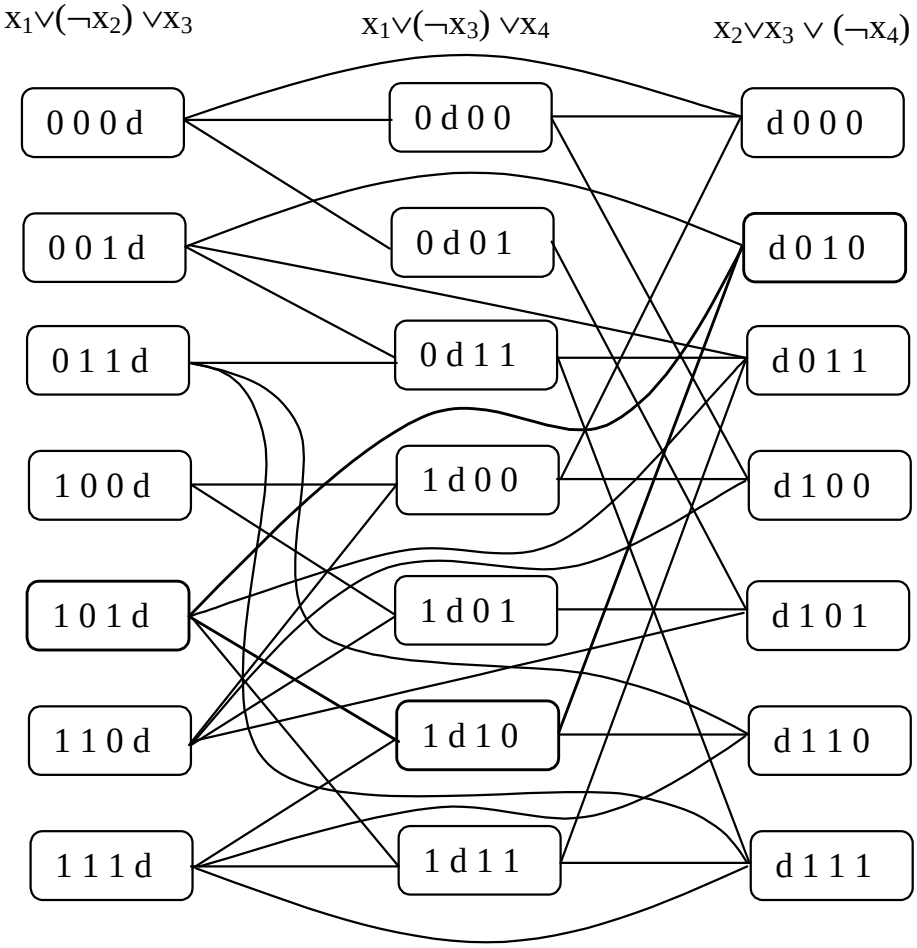
Par exemple, t = d01d0 (n=5) est l'assignation partielle pour laquelle $t(x_2)$ = *faux*, $t(x_3)$ = *vrai*, $t(x_5)$ = *faux*, et les autres variables sont *indéterminées*.

Soit U = $\{x_1,x_2,...,x_n\}$. Deux assignations partielles t et t' sont dites compatibles si : $\forall$ x$\in$U, (t(x) $\neq$ d et t'(x) $\neq$ d) $\Rightarrow$ (t(x) = t'(x)).

Soient $\{c_1, c_2, ..., c_m\}$ les clauses de C.

- L'ensemble de sommets V contient 7 sommets pour chaque clause c_i, notés t_{ij}, j=1,...,7. On peut voir ces sommets comme des assignations partielles avec des valeurs définies uniquement sur les 3 variables de la clause C_i. Sur les 8 ($=2^3$) assignations partielles possibles, on enlève celle qui rend les trois littéraux de c_i faux (et donc la clause elle-même).
- Les arêtes de G connectent toutes les paires d'assignations compatibles.
- $k = m$, le nombre de clauses.

Nous allons montrer que le graphe G construit suivant la méthode ci-dessus a une clique de taille



k **si et seulement si** *F* est satisfiable.

⇒ supposons que G a une clique de taille *k*. Comme *k* = *m* et comme il n'y a pas d'arête joignant deux sommets de la même colonne, la clique est composée nécessairement d'un sommet dans chacune des m colonnes. Comme toutes les assignations partielles correspondantes sont mutuellement compatibles, elles proviennent de la même assignation complète *t*. Et *t* doit satisfaire toutes les clauses de *F* puisque, pour chaque clause, la seule assignation partielle omise dans la colonne correspondante est celle qui rend la clause fausse . Donc *t* satisfait *F* (la rend vraie).

⇐ Supposons que l'on ait une assignation *t* rendant *F* vraie. Alors la restriction de cette assignation aux variables apparaissant dans chaque clause doit correspondre à un sommet du graphe, dans la colonne correspondant à cette clause. Comme ces assignations partielles sont des restrictions de la même assignation complète, elles sont compatibles deux à deux et, par conséquent, forment une clique de taille *k*.

Exemple : Soit $F = (x_1 \vee (\neg x_2) \vee x_3) \wedge (x_1 \vee (\neg x_3) \vee x_4) \wedge (x_2 \vee x_3 \vee (\neg x_4))$. La figure ci-dessous donne le graphe construit selon la méthode exposée précédemment, et une clique a été mise en évidence (en gras) qui correspond à l'assignation x_1=1, x_2=0, x_3=1, x_4=0, qui rend l'expression *C* vraie.

9.7 Le problème d'ordonnancement sur une machine multiprocesseurs est NP-complet

Il s'agit d'ordonnancer des tâches ayant un même temps d'exécution (unitaire) sur un certain nombre de processeurs identiques, sous des contraintes de précédence :

étant donnés un ensemble de tâches $\mathcal{J} = \{J_1,J_2,...,J_n\}$, un graphe orienté acyclique $P(\mathcal{J}, A)$ (graphe de précédence), un entier m (nombre de processeurs) et un entier T (date limite de réalisation), existe-t-il une fonction S (ordonnancement) $S:\mathcal{J} \rightarrow \{1,...,T\}$ telle que :

1. pour tout $j \leq T$, $card(\{J_i / S(J_i) = j\}) \leq m$
2. si $(J_i, J_j) \in A$, alors $S(J_i) < S(J_j)$.

Théorème : le problème d'ordonnancement sur multiprocesseurs est NP-complet

Preuve : le problème est dans NP car il est "facile" de vérifier qu'un ordonnancement est faisable. Nous allons montrer que :

$$\text{CLIQUE} \propto_p \text{ORDONNANCEMENT MULTI\_PROC}$$

Etant donnés un graphe quelconque G=(V,E), et un entier k, on doit construire un ensemble de tâches $\mathcal{J}$, un ordre partiel $P = (\mathcal{J}, A)$ et deux entiers m et T, tels qu'il existe un ordonnancement faisable pour $\mathcal{J}$ si et seulement si G a une clique de taille k. On supposera que G ne contient pas de sommet isolé (ce qui ne nuit pas à la généralité de la preuve puisque les sommets isolés n'influent pas sur la taille des cliques).

1. On définit $\mathcal{J} = V \cup E \cup B \cup C \cup D$; avec B,C,D des ensembles non vides disjoints $B = \{b_1,b_2,...,b_{|B|}\}$, $C = \{c_1,c_2,...,c_{|C|}\}$, $D = \{d_1,d_2,...,d_{|D|}\}$ avec |B|, |C|, |D| tels que :

(1) $m = k + |B| = C_k^2 + |V| - k + |C| = |E| - C_k^2 + |D|$

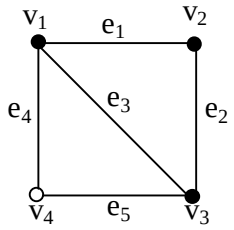
(2) $\min(|B|, |C|, |D|) = 1$

Il n'est pas difficile de vérifier que de tels nombres existent ; m est défini par (1) ; T est fixé égal à 3.

2. Pour la construction de $P = (\mathcal{J}, A)$:

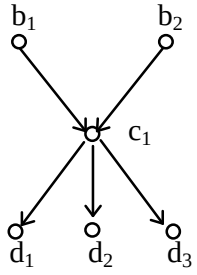
- ajouter à A toutes les arêtes possibles de la forme (b_i, c_j) ou (c_j, d_k),
- ajouter à A les arêtes (v,e) pour tous $v \in V$, $e \in E$ tels que v est une extrémité de e.

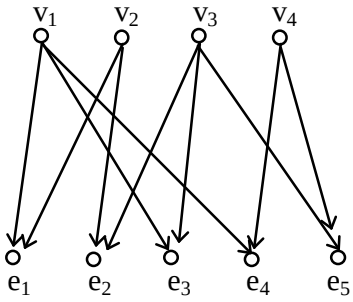
Ceci complète la construction de l'instance de ORDONNANCEMENT MULTI_PROC.
Il s'agit maintenant de prouver qu'il existe un ordonnancement faisable pour $\mathcal{J}$ de



k = 3

T = 3, m = 5
|B| = 2
|C| = 1
|D| = 3





Temps →	1	2	3
machine 1	b_1	c_1	d_1
machine 2	b_2	e_1	d_2
machine 3	v_1	e_2	d_3
machine 4	v_2	e_3	e_4
machine 5	v_3	v_4	e_5

Il s'agit maintenant de prouver qu'il existe un ordonnancement faisable pour $\mathscr{J}$ sous les contraintes A, m et T si et seulement si G a une clique de taille k

Supposons qu'il existe un ordonnancement faisable S. Comme T = 3 et que toutes les tâches de B doivent être exécutées avant celle de C, et celles de C avant celles de D, on a nécessairement :

$S(b_i) = 1$, $S(c_i) = 2$ et $S(d_i) = 3$, pour tout i.

De plus $|\mathscr{J}| = m.T$; donc toutes les machines (au nombre de m) doivent réaliser une tâche à chaque période (pas de temps libre !).

Dans la dernière période, à côté des tâches d, on ne peut exécuter que $m - |D| = |E| - \mathbb{C}_k^2$ tâches correspondantes à des arêtes (les tâches relatives à des sommets ne peuvent être exécutées dans la dernière période, sinon on ne pourrait exécuter les tâches des arêtes dont ces sommets sont extrémités). C'est ici qu'on utilise le fait que G est sans sommet isolé.

On en conclut que :

- les $\mathbb{C}_k^2$ tâches restantes correspondantes à des arêtes doivent être réalisées au cours de la deuxième période,
- les tâches correspondantes aux sommets sont exécutées dans la première période ($m - |B| = k$ d'entre eux) et dans la deuxième période.

De plus les k sommets correspondants aux tâches exécutées dans la première période doivent inclure toutes les extrémités des $\mathbb{C}_k^2$ tâches correspondantes à des arêtes exécutées dans la

deuxième période. Le seul cas où k sommets peuvent comprendre les extrémités de C_k^2 arêtes est une clique de taille k.
L'existence d'un ordonnancement faisable implique donc l'existence d'une clique de taille C.

Réciproquement, si G a une clique C de taille k, on peut construire un ordonnancement faisable S avec :

$S(b_i) = 1$; $S(c_i) = 2$, $S(d_i) = 3$, pour tout i ;
$S(v) = 1$ ssi $v \in C$, $S(v) = 2$ sinon
$S([u,v] = 2$ ssi $u,v \in C$, $S([u,v] = 3$ sinon.

Il est alors immédiat que S est un ordonnancement faisable.

9.8 Le problème du voyageur de commerce (TSP) est NP-complet

Il s'agit de trouver un circuit hamiltonien de coût total inférieur ou égal à une certaine valeur dans un graphe valué.

Théorème : le problème du voyageur de commerce est NP-complet

Preuve : on montre que HC (problème du circuit hamiltonien) est un cas particulier du problème du voyageur de commerce. En effet, étant donné un graphe $G = (V, E)$, on construit une instance de TSP de la manière suivante :

- $|V|$ villes ;
- $d_{ij} = 1$ si $[v_i, v_j] \in E$, $d_{ij} = 2$ sinon ;
- $L = |V|$ le coût.

Il est immédiat qu'il existe un circuit de longueur inférieure ou égale à L si et seulement si il existe un circuit hamiltonien dans G.

10. Résolution des problèmes NP-difficiles

Il est utile de prouver qu'un problème est NP-complet, puisque cela permet de ne pas perdre son temps à rechercher un algorithme polynomial pour le résoudre. Cela justifie aussi l'utilisation d'algorithmes approchés qui, pour les problèmes d'optimisation, permettront de trouver une solution satisfaisante, même sans garantie d'optimalité.
Il existe 4 grandes classes de méthodes générales pour résoudre les problèmes d'optimisation combinatoire NP-difficiles :

- les heuristiques
- les méthodes arborescentes (séparation et évaluation ou *branch and bound*)
- la programmation dynamique
- la programmation linéaire en nombres entiers.

10.1 Les heuristiques

Une heuristique est une **méthode approchée** pour un problème d'optimisation combinatoire, qui a pour but de trouver une solution réalisable, tenant compte de la fonction de coût, mais **sans garantie d'optimalité.**

On oppose les heuristiques aux méthodes exactes qui trouvent toujours l'optimum pourvu qu'on leur en laisse le temps (énumération complète, méthodes arborescentes, programmation dynamique).

Les méthodes exactes ont une complexité exponentielle sur les problèmes NP-difficiles. Sur des problèmes de taille importante, on leur préférera donc des heuristiques.

Types d'heuristiques :

- **méthodes construisant une seule solution** par une suite de choix partiels et définitifs (sans retour en arrière). On retrouve dans cette classe **les méthodes gloutonnes,** quand à chaque itération le choix est le plus avantageux.
- **recherches locales** aussi appelées méthodes de voisinage. On part d'une solution initiale et, par transformations successives, on construit une suite de solutions de coûts décroissants. Le processus s'arrête quand on ne peut plus améliorer la solution courante. Une recherche locale peut être piégée dans un minimum local.
- **recherches globales,** qui sortent des pièges des minimums locaux en construisant une suite de solutions, dans laquelle la fonction de coût peut temporairement remonter.

▲ **problème de l'évaluation des heuristiques :** c'est un problème crucial et souvent difficile. Comme on n'a aucune garantie d'optimalité il est important de pouvoir évaluer la solution trouvée (la distance à l'optimum par exemple). On ne traitera pas cet aspect, dans le cadre de ce cours.

Dans la suite, on illustrera ces différentes classes d'heuristiques pour l'exemple du TSP.

10.2 Heuristiques gloutonnes

On en a déjà vu ! Nous verrons ici les principes de quatre heuristiques gloutonnes appliquées au problème du voyageur de commerce On considère que l'on a un graphe G=(V,E,M) simple complet et valué de n sommets, défini par une matrice carrée symétrique M ($M[i, j]$ désigne le coût de l'arête (i, j)).
Pour éviter le problème d'existence, on suppose que le graphe est complet, de façon que toute permutation des sommets définisse un cycle hamiltonien. Même avec cette hypothèse simplificatrice, le problème reste NP-difficile.

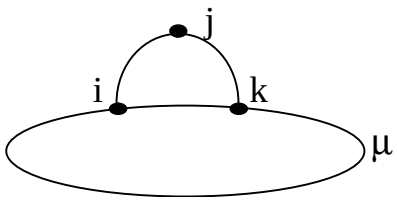
10.2.1 Plus proche voisin (PPV)

PPV

- partir d'un sommet quelconque, par exemple le sommet 1
- tant qu'il reste des sommets libre faire
  connecter le dernier sommet atteint au sommet libre le plus proche
- relier le dernier sommet au sommet 1.

C'est l'heuristique la plus simple. Elle peut être mise en oeuvre en $\Theta(n^2)$. **Heuristiques d'insertion (PLI, PPI, MI)**

Elles choisissent la position d'insertion d'un nouveau sommet dans un cycle. On définit la distance d'un sommet à un cycle, la longueur de l'arête la plus courte joignant ce sommet à un sommet du cycle.



PLI (Plus Lointaine Insertion)

- partir d'un cycle μ réduit à une boucle sur le sommet 1 (par exemple)
- tant qu'il y a des sommets libres faire
  choisir le sommet libre j le plus loin de μ
  chercher la position d'insertion entre 2 sommets i, k du cycle μ minimisant l'accroissement du coût $\Delta M = M_{ij} + M_{jk} - M_{ik}$

PPI (Plus Proche Insertion)

- partir d'un cycle μ réduit à une boucle sur le sommet 1 (par exemple)
- tant qu'il y a des sommets libres faire
  choisir le sommet libre j le plus proche de μ
  chercher la position d'insertion entre 2 sommets i, k du cycle μ minimisant l'accroissement du coût $\Delta M = M_{ij} + M_{jk} - M_{ik}$

MI (Meilleure Insertion)

- partir d'un cycle μ réduit à une boucle sur le sommet 1 (par exemple)
- tant qu'il y a des sommets libres faire
  pour tous les sommets libres j chercher la position d'insertion entre 2 sommets i, k du cycle μ minimisant $\Delta M = M_{ij} + M_{jk} - M_{ik}$
  insérer le sommet qui minimise l'accroissement du coût

En moyenne ces heuristiques sont très bonnes. MI est en $\Theta(n^3)$ et a le meilleur comportement moyen sur des graphes quelconques. PPI et PLI sont en $\Theta(n^2)$.

10.3 Recherches locales (k-OPT)

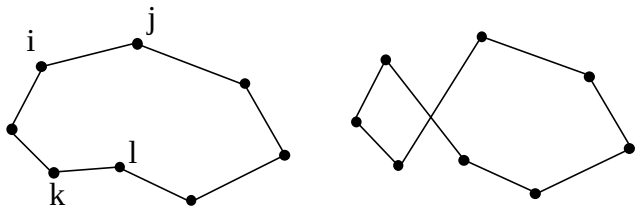
Soit un problème d'optimisation combinatoire (F, c), où F est l'ensemble des solutions possibles et c la fonction de coût. Un algorithme de recherche locale a la structure générale suivante, avec $V(s)$ un **voisinage** de la solution s à définir :

$s \leftarrow$ solution initiale
$z \leftarrow c(s)$
tant qu'il existe s' dans $V(s)$ avec $c(s') < z$ faire
    $s \leftarrow s'$
    $z \leftarrow c(s)$

Dans le cas du TSP, on part d'un cycle initial construit par une des heuristiques précédentes, puis on construit une suite de cycles de coûts décroissants. Chaque cycle résulte du précédent par une transformation plus ou moins simple. L'ensemble des transformations possibles pour un cycle définit un *voisinage.*

Le voisinage le plus connu pour le TSP est appelé k-OPT, il consiste à enlever k arêtes et à reformer un cycle en ajoutant k autres arêtes. Le voisinage d'un cycle a donc N^k éléments

2-OPT
C'est le cas le plus simple : on remplace deux arêtes non-consécutives (i, j) et (k,l) par (i,k) et (j,l), à condition que le coût du cycle diminue.
2-OPT ne convient pas pour le cas de graphes orientés.



cycle de coût Z cycle de coût $Z + M_{il} + M_{jk} - M_{ij} - M_{kl}$

Pour 3-OPT et 4-OPT, le gain obtenu est faible par rapport à l'accroissement de la complexité de l'algorithme. Ces recherche locales sont meilleures en moyenne que les méthodes gloutonnes, mais leur comportement au pire est le même.

10.4 Méthodes de recherche globale

10.4.1 Recuit simulé (simulated annealing)

Cette méthode a été inventée par des physiciens en 1983. Le nom provient de l'analogie avec le recuit des métaux (*annealing*) en métallurgie : un métal refroidi trop vite présente de nombreux défauts microscopiques ($\rightarrow$ minimum local d'un problème d'optimisation), si on le refroidit lentement sa structure est alors bien ordonnée ($\rightarrow$ minimum global d'un problème d'optimisation).

Autre analogie :
- pierre lâchée dans un paysage montagneux, elle se stabilise dans le premier creux trouvé (recherche locale)
- balle lâchée dans le même paysage, elle peut rebondir et contourner des obstacles, elle s'arrêtera probablement plus bas que la pierre

T doit être diminuée très lentement à chaque itération (il faudra plusieurs milliers d'itérations pour que l'algorithme soit efficace). Le seuil fixé pour l'arrêt de l'agorithme (ε) est proche de zéro.

Recuit simulé (schéma général)

- donner une température T (réel arbitraire)
- donner une solution réalisation s
- répéter
  - tirer au sort une transformation c'est-à-dire une solution réalisable s' du voisinage de s
  - calculer la variation de coût Δf
  - si $\Delta f \leq 0$: le coût diminue, effectuer la transformation ($s \leftarrow s'$)
  - si $\Delta f > 0$: le coût augmente, c'est un rebond qu'on pénalise d'autant plus que la température est basse et que Δf est grand :
    calculer une probabilité d'acceptation $a = e^{\frac{-\Delta f}{T}}$
    tirer au hasard un nombre $p \in [0,1]$
    si $p \leq a$, la transformation est acceptée ($s \leftarrow s'$)
    sinon ($p > a$) la transformation est rejetée
  - diminuer T

jusqu'à $T \leq \varepsilon$

10.4.2 Méthodes taboues

Les recherches taboues datent de 1985 (Glover), elles n'ont aucun caractère stochastique, et paraissent meilleures, à temps d'exécution égal, que le recuit simulé.
Trois principes de base :

- à chaque itération, on examine complètement le voisinage de la solution actuelle et on prend la solution s' qui donne la variation de coût la plus petite (même si le coût augmente).
- on s'interdit de revenir sur une solution visitée dans un passé trop proche grâce à la gestion d'une liste taboue (qui contient les inverses des transformations déjà réalisées).
- on conserve la meilleure solution trouvée en cours de route (ce n'est pas forcément la dernière) et on stoppe au bout d'un nombre donné d'itérations.

Deux points délicats : l'obtention d'une solution de départ (plus on est proche de l'optimum, plus l'algorithme sera efficace), et la longueur de la liste taboue (voir l'exemple traité en TD).

10.4.3 Comparaisons expérimentales des heuristiques

Heuristique	**non euclidiens**		**euclidiens**	
	coût	**durée (s)**	**coût**	**durée (s)**
Plus proche voisin	4613	0.04	6958	0.04
Plus proche insertion	6699	0.11	6682	0.10
Plus lointaine insertion	7116	0.11	6173	0.10
Meilleure insertion	6233	1.44	6491	1.28
2-OPT	2862	0.99	5703	1.13
Recuit	3691	16.36	5915	17.39
Tabou	2666	8.58	5672	8.59