931 — Schémas algorithmiques. Exemples et applications.

Motivations : Trouver des paradigmes de programmation qui permettront d'utiliser les mêmes procédés pour résoudre des algorithmes ou du moins améliorer leur complexité.

(I) Diviser pour régner

Butt 1 : traiter des instances plus petites du même problème puis les combiner pour avoir la solution cherchée.

Ex 2 : Algorithme de DICHOTOMIE

entrée : T est trié, $0 \leq a < b \leq n$, $n = T.\text{taille}$

sortie : indice d'une occurrence de x si $x \in T$, NIL sinon.

complexité : $O(\lg(n))$

```
Dicho(T, a, b, x):
si a=b et T[a]=x:
    renvoie a
si a=b et T[a]≠x:
    renvoie NIL
si x > T[⌊(a+b)/2⌋]:
    renvoie dicho(T, ⌊(a+b)/2⌋+1, b, x)
sinon
    renvoie dicho(T, a, ⌊(a+b)/2⌋, x)
```

Thm 3 (MASTER THEOREM) : Soit $T : \mathbb{N} \to \mathbb{R}_+$ fonction croissante à partir d'un certain rang.
$n_0 \geq 1$, $b \geq 2$ des entiers
$k \geq 0$, $a > 0$, $c > 0$, $d > 0$ des réels
tels que $T(n_0) = d$
$T(n) = aT(\frac{n}{b}) + cn^k$ pour $n = n_0 b^p$, $p \in \mathbb{N}$

Alors $T(n) = \begin{cases} (A)\ (n^k) & \text{si } a < b^k \\ (B)\ (n^k \log_b(n)) & \text{si } a = b^k \\ (C)\ (n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$

DEV (T)

Appli 4 : TRI FUSION en $O(n\log_2(n))$ — $a=2$ $b=2$ $k=1$ $n_0=1$ $d=1$ $c=1$

Rmq 5 : Ce théorème 3 ne permet pas de donner la complexité de tout algorithme utilisant le paradigme diviser pour régner

Ex 6 : Le TRI RAPIDE ne peut pas être traité par le Master Thm car on divise notre entrée en des parts de tailles qui diffèrent.

Ex 7 : La multiplication de matrices par la METHODE de STRASSEN permet de passer de $O(n^3)$ (algo naïf) à $O(n^{\log_2(7)})$ (Strassen) $(\log_2(7) \simeq 2{,}81)$

Limite 8 : Tous les algorithmes qui utilisent diviser pour régner ne sont pas forcément efficace.

Ex 9 : Pour calculer le $n^{ième}$ nombre de FIBONACCI, calculer récursivement le $(n-1)^{ième}$ et le $(n-2)^{ième}$ et les ajouter n'est pas efficace.

```
FiBo(n)                                  (1)
si n=0 ou n=1
    renvoie 1
sinon renvoie
        FiBo(n-1)
    + FiBo(n-2)
```

Complexité : $O(a^n)$

```
FiBo(n)                                  (2)
S ← 1
T ← 1
pour i allant de 2 à n
    U ← S
    S ← T
    T ← S+U
renvoie T
```

Complexité : $O(n)$

Notation 14bis : $X_i = \langle x_1, \dots, x_i \rangle$ et $Y_j = \langle y_1, \dots, y_j \rangle$

(II) Programmation dynamique

But 10 : mémorise les solutions des sous problèmes (mémoïsation) pour résoudre le problème global.

Ex 11 :

```
FiBo(n)                                   (3)
    T = [1,1,0,...,0] (de taille n+1)
    pour i allant de 2 à n
        T[i] ← T[i-1] + T[i-2]
    renvoie T[n]
```

complexité : $O(n)$ (temporelle) ; complexité : $O(n)$ (spatiale)

Rmq 12 : La mémoïsation nécessite de la mémoire en plus mais ne recalcule pas de sous problèmes déjà calculés.

Ex 13 : Algorithme COCKE YOUNGER KASAMI résolvant le problème du mot polynomialement en utilisant la programmation dynamique (DEV (2))

Ex 14 : PLUS LONGUE SOUS SEQUENCE COMMUNE

Soient deux séquences $X = \langle x_1, \dots, x_m \rangle$ et $Y = \langle y_1, \dots, y_n \rangle$

on veut trouver le plus grand $k \in \{0, \dots, \min(n,m)\}$ tel qu'il existe $i_1, \dots i_k \in \{1, \dots, m\}$ croissant strictement et $j_1, \dots j_k \in \{1, \dots, n\}$ croissant strictement et $x_{i_\ell} = y_{j_\ell}$ $\forall \ell \in \{1, \dots, k\}$ on note $z_\ell = x_{i_\ell}$

on dira que $\langle z_1, \dots, z_k \rangle$ est une plus longue sous séquence commune à X et Y, notée PLSC(X,Y)

Thm 15 : Soient X, Y deux séquences et Z une PLSC(X,Y)

1) Si $x_m = y_n$ alors $z_k = x_m = y_n$ et Z_{k-1} est une PLSC(X_{m-1}, Y_{n-1})
2) Si $x_m \neq y_n$ alors $z_k \neq x_m \Rightarrow$ Z est une PLSC(X_{m-1}, Y)
3) Si $x_m \neq y_n$ alors $z_k \neq y_n \Rightarrow$ Z est une PLSC(X, Y_{n-1})

Corollaire 16 : En notant c[i,j] la longueur d'une PLSC(X_i, Y_j)

on a
$$c[i,j] = \begin{cases} 0 & \text{si } i=0 \text{ ou } j=0 \\ c[i-1,j-1]+1 & \text{si } i,j>0 \text{ et } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{si } i,j>0 \text{ et } x_i \neq y_j \end{cases}$$

annexe

Ex 17 : Pour calculer la distance d'édition entre deux mots, on peut utiliser la programmation dynamique

Ex 18 : L'algorithme de FLOYD-WARSHALL utilise le paradigme de la programmation dynamique.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{si } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)}) & \text{si } k \geq 1 \end{cases}$$

```
FLOYD-WARSHALL(W) graphe pondéré sans cycle de poids négatif
    n = W.lignes
    D(0) = W
    pour k allant de 1 à n
        soit D(k) nouvelle matrice n x n
        pour i allant de 1 à n
            pour j allant de 1 à n
                d_ij(k) = min(d_ij(k-1), d_ik(k-1), d_kj(k-1))
    renvoie D(n)
```

Complexité : $O(n^3)$ (temporelle)

PLUS COURT CHEMIN ENTRE TOUTES LES PAIRES DE SOMMETS

III Algorithmie gloutonne

Bubly: Faire des choix localement optimaux pour obtenir une solution globalement optimale.

Rmq 20: Attention tout algorithme glouton ne redonne pas forcément une solution globalement optimale

Ex 21 : L'algorithme glouton de rendu de monnaie en un minimum de pièces n'est pas toujours optimale (cela dépend de la valeur des pièces)

Ex 22 : Les algorithmes de PRIM et KRUSKAL sont deux algorithmes gloutons qui renvoient des solutions optimales.

ARBRE COUVRANT MINIMAL

```
KRUSKAL(G=(S,A), w)
    E = ∅
    pour v ∈ S
        créate(v)
    trier A par ordre croissant de poids w
    pour (u,v) ∈ A (pris par ordre croissant)
        si FIND(u) ≠ FIND(v)
            E = E ∪ {(u,v)}
            UNION(u,v)
    renvoie E
```

G non orienté connexe
$w : A \to \mathbb{R}$

complexité
$O(|A| \log |A| + |A| \alpha(|S|) + |S|)$
(temporelle)

utilisation de UNION FIND

ARBRE COUVRANT MINIMAL

```
PRIM(G=(S,A), w, r)
    pour u ∈ S
        u.clé = ∞
        u.π = NIL
    r.clé = 0
    F = S
    tant que F ≠ ∅
        u = Extraire_min(F)
        pour v ∈ Adj(u)
            si v ∈ F et w(u,v) < v.clé
                v.π = u
                v.clé = w(u,v)
    renvoie π
```

G non orienté connexe
$w : A \to \mathbb{R}$
$r \in S$

complexité :
$O(|S| \log |S| + |A| \log |S|)$
(temporelle)

utilisation d'une file de priorité

Annexe :

x_i \ y_j		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4