926 Analyse d'algorithme : Complexité . Exemples

<u>Introduction</u> : on veut pouvoir ~~résoudre~~ résoudre un problème P, quelle que soit les conditions de départ, en suivant un schéma de résolution prédéfini et reproductible par n'importe qui.

<u>Def 1</u> : Un algorithme décrit un traitement sur un certain nombre, fini, de données.

- Un algorithme est la composition d'un nombre fini d'étape, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :
  - définie de façon rigoureuse et non ambigüe.
  - effective (pouvant être réalisée par une machine).
- Quelle que soit la donnée sur laquelle on travaille un algorithme doit toujours se terminer après un nombre fini d'opérations

<u>Objectifs</u> : Exhiber des critères **de comparaison** entre algorithme qui sont indépendants de la machine qui les exécute.

<u>Def 2</u> : Soit A un algorithme. On choisit un ensemble d'opération fondamentales parmis les opérations pouvant être effectuées par l'algo. Pour une instance x du problème P, on définit $C(x)$ comme le nombre d'opérations élémentaires effectuées par l'algo A sur l'entrée x.

<u>Ex</u> : Recherche séquentielle d'un nom dans un annuaire contenant n noms.
opération fondamentale : comparaisons.
→ si le nom recherché est en position i, alors $C(x) = i$.

Pour s'exécuter, l'algorithme A doit manipuler les données du problème, ainsi que des données auxiliaires. On suppose que **l'on stock les** informations dans des blocs mémoire.

<u>Def 3</u> : Soit x une instance, on note $C_S(x)$ le nombre de bloc mémoires utilisés par l'algorithme sur l'entrée x.

<u>Ex</u> : Recherche séquentielle d'un élément dans un tableau de taille n : il faut $n+1$ blocs mémoires.

II - <u>Complexité en temps</u>

<u>Def 4</u> : Soit D_n l'ensemble des données de taille n traitées par l'algorithme A, on définit les complexités

Au pire : $C_{max}(n) = \max_{x \in D_n} C(x)$

$C_{min}(n) = \min_{x \in D_n} C(x)$

$C_{moy}(n) = \sum_{x \in D_n} p_n(x)\, C(x)$

où p_n est une distribution de proba sur D_n.

En pratique on ne s'intéresse pas à l'expression exacte de ces complexité, on veut simplement une majoration asymptotique.

Defs : Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$, on dit que :

- $f = O(g)$ ssi $\exists c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, \quad f(n) < c g(n)$.
- $f = \Theta(g)$ ssi $f = O(g)$ et $g = O(f)$.

Exemple $\boxed{\text{DVPT}}$ **Comparaison du Tri rapide et du Tri fusion.**

2) Techniques de calculs.

• Lors de l'utilisation d'algorithmes récursifs, on est souvent amenés à résoudre des relations de récurrences sur les complexités.

Exemple : Recherche dichotomique dans un tableau trié.

On a deux outils

Th (Général) Soient $a \geq 1$ et $b > 1$ deux ctes, $f(n)$ une fonction et $T(n)$ défini pour les entiers positifs par la récurrence

$$T(n) = a T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$$

alors $T(n)$ est asymptotiquement bornée de la façon suivante :

1) Si $f(n) = O(n^{\log_b(a) - \varepsilon})$ pour un certain $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b(a)})$

2) Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log(n))$

3) Si $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$ pour un certain $\varepsilon > 0$, et si $a f\left(\frac{n}{b}\right) \leq c f(n)$ pour un certain $c < 1$ et pour n suffisamment grand, alors $T(n) = \Theta(f(n))$

• Utilisation de série génératrice : Lorsque l'on tombe sur une relation de récurrence du type

$$C(n) = C(n-a) + C(n-b)$$

on multiplie par z^n des deux côtés, et on somme pour toutes les valeurs de n.

$$\sum_{n \in \mathbb{N}} C(n) z^n = z^a \sum C(n-a) z^{n-a} + z^b \sum_{n \in \mathbb{N}} C(n-b) z^{n-b}$$

Puis on résoud l'équation $f(x) = x^a f(x) + x^b f(x)$

$C(n)$ est donc le n-ième coefficient de la série de Taylor de f.

• Analyse Amortie : au lieu de majorer chaque opération individuellement on s'intéresse au coût moyen de chaque opération dans une suite de n opérations. ↳ $\triangle$ c'est le cas le pire

- Méthode de l'agrégat : Si une suite de n opérations coûte $T(n)$, alors le coût amorti de chaque opération est $\frac{T(n)}{n}$

Ex : Incrémentation d'un compteur binaire de k bits.

Entrée : Tableau A.

```
i=0
tant que i < A.longueur et A[i] = 1
    A[i] = 0
    i = i+1
si i < A.longueur
    A[i] = 1
```

Sur une suite de n opérations, en fait, le i-ème bits n'est modifié, au plus, que $\left\lfloor \frac{n}{2^i} \right\rfloor$ fois, ce qui donne : $T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{+\infty} \frac{1}{2^i} < 2n$

Donc le coût amorti est $\frac{T(n)}{n} = O(1)$ (Au lieu de $O(k)$).

- **Méthode comptable** : si certaines opérations ne peuvent pas être effectuées avant d'autres opérations, alors on surévalue les premières et on sous-évalue les dernières :

Ex : Pile : empiler 1 ⟶ empiler 2
dépiler 1 ⟶ dépiler 0
vider k ⟶ vider 0

- **Méthode du potentiel** : Dans cette méthode, on alloue un potentiel de travail à la structure de donnée. Le coût de chaque opération est donc le coût réel + différence de potentiel.

Ex : Potentiel Φ = longueur de la pile ; D_i = structure de donnée après i opérat°

Empiler $= 1 + \Phi(D_n) - \Phi(D_{n-1}) = 1+1 = 2$ ✓

vider $= k + \Phi(D_n) - \Phi(D_{n-1}) = k + 0 - k = 0$ ✓

3) Complexité moyenne

Dans un algorithme qui doit être exécuté de nombreuses fois sur un ensemble d'entrée, il est plus réaliste de calculer la complexité moyenne.

Deux types de problèmes :

- Quelle distribution de proba utiliser ?

Ex : algorithme de traduction mot à mot. Pour traduire du Français vers une autre langue, on aimerait que la probabilité d'apparition de "le" soit plus élevée que celle de "mouche". (sauf si c'est un livre de cuisine ...).

- Etant donné une distribution de proba, construire un algo qui minimise la complexité moyenne.

Ex : DVPT : construction des ABRO

(Et je crois qu'on peut parler de table de Hachage ?).

III. Le bon compromis espace/temps.

L'étude de la complexité en espace est plus proche de la machine que celle de la complexité en temps. La principale motivation est que l'accès en mémoire secondaire est beaucoup plus lent que l'accès à la mémoire principale. Une grosse complexité en espace peut ralentir un algorithme en pratique.

Mais on peut aussi diminuer la complexité en temps en augmentant la complexité en espace. Cette astuce est utilisée en programmation dynamique.

Ex : Fibonacci récursif : $C(n) = O(\varphi^n)$

Fibonacci programmation dynamique : $C(n) = O(n)$.

- Construction d'ABRO ✓ (encore !).

Je craque ... :@

Biblio : Froidevaux (Définitions) et (ABRO)
Cormen (Analyse amortie)
Flajolet (Série Génératrices)

return 0;;

Tri Rapide - Tri Fusion [Cormen]

Algorithmes

```
FUSION (A,p,q,r)
  n1 ← q-p+1
  n2 ← r-q
  Creer-Tableau L [1, n1+1]
  Creer-Tableau D [1, n2+1]
  Pour i de 1 à n1   L[i] ← A[p+i-1]
  Pour j de 1 à n2   D[j] ← A[q+j]
  L[n1+1] ← ∞   ;   D[n2+1] ← ∞
  i := 1        ;   j := 1.
  Pour k de p à r faire
    Si L[i] ≤ R[j]
    Alors A[k] ← L[i]
          i ← i+1
    Sinon A[k] ← R[j]
          j ← j+1
```

```
PARTITION (T, p, r)
  x ← A[r]   ;   i ← p-1
  Pour j de p à r-1 faire
  | Si A[j] ≤ x Alors
  | | i ← i+1
  |   A[i] ⟷ A[j]
  A[i+1] ⟷ A[r]
  Retourner i+1.
```

```
TRI-FUSION (T, p, r)
  Si p<r Alors
    q ← ⌊(p+r)/2⌋            (* Coût de séparation nul *)
    TRI-FUSION (T, p, q)
    TRI-FUSION (T, q+1, r)
    FUSION (T, p, q, r).
```

```
TRI-RAPIDE (T, p, r)
  Si p<r Alors
    q ← PARTITION (T, p, r)
    TRI-RAPIDE (T, p, q-1)
    TRI-RAPIDE (T, q+1, r).
                              (* Coût de Fusion nul *)
```

Complexité en temps

Ici, on ne compte que le nombre de comparaisons.

Pour le TRI-FUSION : Puisque FUSION fait exactement n comparaisons sur un tableau de taille n.

On a $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n$ $\qquad T(0) = T(1) = 1$.

On résout la récurrence pour $n = 2^{k+1}$.

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1}$$

soit $\dfrac{T(2^{k+1})}{2^{k+1}} = \dfrac{T(2^k)}{2^k} + 1$ $\qquad$ Par récurrence: $\dfrac{T(2^k)}{2^k} = k$

Donc, en supposant que la complexité est une fonction croissante de n:

$$\boxed{T_F(n) = \Theta(n \log n).}$$

Pour le TRI-RAPIDE : PARTITION effectue $n-1$ comparaisons sur un tableau de taille n. Par contre, le coût de TRI-RAPIDE dépend de l'entrée.

Au PIRE : L'un des deux tableaux obtenus est vide, à chaque appel récursif :

$$C_p(n+1) = n + C_p(n) \quad \text{soit} \quad C_p(n) = \Theta(n^2)$$

En MOYENNE : Si le pivot est en j-ième position après PARTITION :

l'algorithme s'appelle récursivement sur des entrées de taille $j-1$ et $n-j$.

On suppose que toutes les permutations de l'entrée sont équiprobables.

La probabilité que le pivot soit en j-ième position est donc de $\frac{1}{n}$, soit

$$C_{moy}(n) = n-1 + \frac{1}{n}\sum_{j=1}^{n}\left(C(j-1) + C(n-j)\right), \qquad C(0) = C(1) = 0$$

$$= n-1 + \frac{2}{n}\sum_{j=1}^{n-1} C(j).$$

Soit $$n\left(C(n) - n + 1\right) = 2\sum_{j=1}^{n-1} C(j)$$

Et, au rang $n-1$: $$(n-1)\left(C(n-1) - n + 2\right) = 2\sum_{j=1}^{n-2} C(j).$$

On soustrait : $$nC(n) - (n-1)C(n-1) - 2n + 2 = 2C(n-1)$$

Soit $$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \leq \frac{C(n-1)}{n} + \frac{2}{n+1}$$

Par récurrence : $$C(n) \leq (n+1)\sum_{k=1}^{n}\frac{2}{k+1} = 2(n+1)\left(H_{n+1} - 1\right) \underset{n\to+\infty}{\sim} 2n\ln n$$

Donc $\boxed{C(n) = \Theta(n \log n).}$ mais la constante est de l'ordre de 1,38.

Contre 1 pour le TRI-FUSION. Celui-ci a donc l'air plus efficace...

Mais on n'a pas pris en compte le coût des affectations dans les tableaux :

$2n+2$ pour FUSION contre $2\cdot\left(\frac{n-1}{2}\right) + 2 = n+1$ en moyenne pour PARTITION.

Cela ne modifie pas la complexité asymptotique mais relativise l'importance de nos constantes...

Enfin il faut prendre en compte la complexité en espace, mais il est difficile de la calculer précisément...

En pratique c'est le TRI-RAPIDE qui donne les meilleures performances.

Arbre Binaire de Recherche Optimal [Cormen]

On veut construire une structure de donnée gérant un dictionnaire
et l'ensemble des clés est statique. Le but est de minimiser le coût
moyen d'une recherche (la seule opération du dictionnaire).

Soit $k_1 < \dots < k_n$ les clés du dictionnaire. L'univers des clés possibles n'étant
pas limité à $\{k_1 \dots k_n\}$, on introduit des clés factices $d_0 \dots d_n$ qui signifient :
" l'entrée se trouve entre k_i et k_{i+1} ".

On connait les probabilités que la recherche renvoie k_i (tableau $p[1..n]$)
ou d_j (tableau $q[0..n]$) : on a $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$.

- On va utiliser un ABR, qui donne un coût au pire de $\Theta(\log n)$ pour l'opération RECHERCHER. La complexité moyenne d'une recherche est alors :

$$\boxed{C_T = \sum_{i=1}^{n} (\mathrm{prof}_T(k_i)+1)\, p_i + \sum_{j=0}^{n} (\mathrm{prof}_T(d_j)+1)\, q_j}$$

$\boxed{\text{Trouver } T \text{ tq } C_T \text{ minimal ?}}$

- Les clés factices sont des feuilles (fin de recherche). Le nombre d'ABR que l'on peut construire est donc le nombre d'arbres binaires à n noeuds, qui est exponentiel.
- L'algorithme naïf d'énumération est donc inefficace, on va faire appel à la Progra Dynamique

1) Sous-Problèmes

Un sous-arbre d'un ABR contient les clés $k_i \dots k_j$ et $d_{i-1}, \dots d_j$ $\begin{cases} i,j \in [\![1,n]\!] \\ i \leq j+1 \end{cases}$

Dans un ABRO, tout sous-arbre doit être optimal. Si sa racine est k_r, $i \leq r \leq j$,
alors il est de la forme $T' = $ (k_r)

$k_i \dots k_{r-1}$ / $d_{i-1} \dots d_{r-1}$ $\qquad$ $k_{r+1} \dots k_j$ / $d_r \dots d_j$

On note $e[i,j] = \sum_{\ell=i}^{j} (\mathrm{prof}_{T'}(k_\ell)+1)\, p_\ell + \sum_{\ell=i-1}^{j} (\mathrm{prof}_{T'}(d_\ell)+1)\, q_\ell$.

et $w[i,j] = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell$ la probabilité que la recherche s'achève dans T'.

2) Formule de récurrence

On a : $\boxed{w[i,j] = w[i,j-1] + p_j + q_j}$

et $e[i,j] = w[i,j] + \sum_{\text{Gauche}(T')} \mathrm{prof}_{T'}(k_\ell)\, p_\ell + \sum_{\text{Gauche}(T')} \mathrm{prof}_{T'}(d_\ell)\, q_\ell + \sum_{\text{Droit}(T')} \mathrm{prof}_{T'}(k_\ell)\, p_\ell + \sum_{\text{Droit}(T')} \mathrm{prof}_{T'}(d_\ell)\, q_\ell$

soit $\boxed{e[i,j] = w[i,j] + e[i,r-1] + e[r+1,j]}$ où k_r est la racine de T'

3) Algorithme

```
ABRO (p[1..n], q[0..n])
    Creer_Tableau  e[1..n+1, 0..n]
    Creer_Tableau  w[1..n+1, 0..n]
    Creer_Tableau  r[1..n, 1..n]
    Pour i de 1 à n+1 faire
        e[i, i-1] := q_{i-1}
        w[i, i-1] := q_{i-1}
    Pour l de 1 à n faire
        Pour i de 1 à n+1-l faire
            j := i+l-1
            e[i,j] := ∞
            w[i,j] := w[i, j-1] + p[j] + q[j]
            Pour r de i à j faire
                t := e[i, r-1] + e[r+1, j] + w[i,j]
                Si t < e[i,j] alors
                    e[i,j] := t
                    r[i,j] := r
    Retourner e et r
```

($r[i,j]$ est la racine du sous-ABRO $\left(\begin{smallmatrix} k_i \dots k_j \\ d_{i-1} \dots d_j \end{smallmatrix}\right)$)

Correction

- Terminaison : boucles Pour
- Invariant de boucle :

" $\frac{e[i,j]}{w[i,j]}$ est le coût moyen optimal d'une recherche dans un ABR de clés $k_i \dots k_j$ et $d_{i-1} \dots d_j$, de probabilités $\frac{1}{w[i,j]} p[i..j]$ et $\frac{1}{w[i,j]} q[i-1..j]$ "

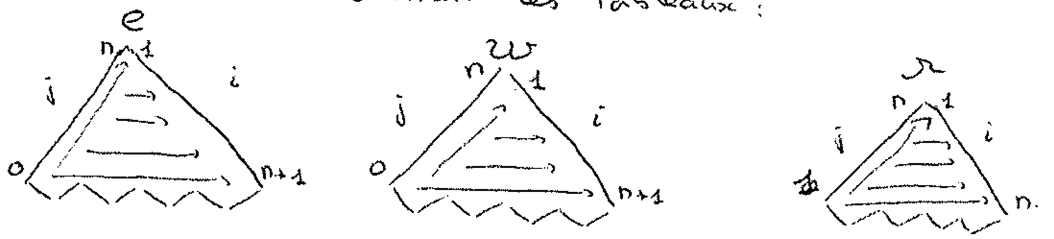
$\Rightarrow C_T = e[1..n]$

4) Complexité

- Temporelle au pire et en moyenne : dans les deux cas la boucle sur r est en $\Theta(1)$.

  Donc $\boxed{C(n) = \Theta(n^3)}$

- Spatiale : on construit les tableaux :



$= \boxed{\Theta(n^2)}$

- Optimisation Spatiale :

  Pour w, on peut se contenter d'un tableau $[1..n+1]$

  Pour les deux autres tableaux, on doit conserver toute l'information.

5) Exemple : Cf [Cormen], p 348 et 353.

Algorithme de Dijkstra : Complexité [Cormen]

Donnée : $G = (S, A)$ graphe, $W : A \to \mathbb{N}$ fonction de poids, $s \in S$ source.

But : Calculer $d(s,v)$ pour $v \in S$ et donner un chemin de s à v de longueur minimale.

On note $S = [1..n]$, on va construire $d[1..n]$ et $\pi[1..n]$ à l'aide d'une file de priorité min F d'ensemble $[1..n]$ dont les clés sont stockées dans $d[1..n]$.

On utilise les opérations INSERER(F, v, clé) $\rightsquigarrow$ $d[v] \leftarrow$ clé, place v dans F

EXTRAIRE_MIN(F) $\rightsquigarrow$ Renvoie $v \in F$ tq $d[v]$ minimal dans F, supprime v de F.

DIMINUER_CLE(F, v, clé) $\rightsquigarrow$ Si $v \in F$, $d[v] \leftarrow$ clé, réorganise F

et la fonction :

```
RELACHER (u, v, W) :
    Si d[v] > d[u] + W(u,v) Alors
        DIMINUER_CLE (F, v, d[u]+W(u,v))
        π[v] ← u.
```

Dans

```
DIJKSTRA (G, W, s)
    Creer_Tableau π[1..n] à NIL
    Creer_Tableau d[1..n]
    F := ∅. Pour v ∈ S Faire INSERER(F, v, ∞)
    DIMINUER_CLE (F, s, 0)
    Tant Que F ≠ ∅ Faire
        u ← EXTRAIRE_MIN (F)
        Pour v ∈ Adj[u] Faire
            RELACHER (u, v, W).
```

Terminaison : Le "variant" de la boucle Tant que est $|F|$

Correction : Demander à Rémi (leçon 901)

Complexité (en temps au pire).

On note $I(n)$, $E(n)$ et $D(n)$ les complexités de INSERER, EXTRAIRE_MIN, DIMINUER_CLE

Alors $C(n) = 2n+1 + \sum_{k=0}^{n-1} I(k) + \sum_{k=1}^{n} \Big(E(k) + |Adj| (2 + D(k)) \Big)$

$\leq n.I(n) + n E(n) + 2A(2 + D(n)) + O(n)$

Implémentation F	$I(n)$	$E(n)$	$D(n)$	Matrice d'adjacence	Liste d'ajacence
Tableau	1	n	1	$n + n^2 + n^2 \times 3 + O(n) = \Theta(n^2)$	$\Theta(n^2 + A) = \Theta(n^2)$
Tas Min	$O(1)$ (Amorti)	$\log n$	$\log n$	$O(n) + n \log n + n^2(2 + \log n) = \Theta(n^2 \log n)$	$\Theta((n+A)\log n)$

Conclusion : Dans les deux cas, on préfèrera une liste d'adjacence pour implémenter notre graphe, mais le gain n'est pas substantiel... Si le graphe est peu dense (par exemple si on sait borner le degré des sommets du graphe), l'implémentation de F par tas_min est beaucoup plus efficace, mais elle est moins efficace pour un graphe très dense!