INF4705 — Analyse et conception d'algorithmes

CHAPITRE 3 : ANALYSE D'ALGORITHMES

Comment mesurer l'efficacité d'un algorithme ?

3.1 Trois approches

Approche empirique (a posteriori) : On l'implante puis on l'exécute sur différents exemplaires du problème, prenant soin de mesurer la quantité de ressource(s) consommée(s).

+ évaluation précise de l'efficacité
− nécessite l'implantation de l'algorithme
− dépend du talent du programmeur, du matériel (cadence du processeur, mémoire vive, cache), du logiciel (langage, compilateur)
− restreint à un nombre relativement petit d'exemplaires, surtout de taille petite à moyenne

Approche théorique (a priori) : On exprime mathématiquement la consommation de ressource(s) en fonction de la taille des exemplaires. Cette fonction possède quelques paramètres non spécifiés.

+ pas besoin d'implanter l'algorithme
+ indépendant du programmeur, du langage et de l'ordinateur
+ s'applique à l'ensemble des exemplaires de tailles variées
− ne donne qu'un ordre de grandeur de l'efficacité (à une constante multiplicative près)

Approche hybride : On détermine la fonction par l'approche théorique puis ses paramètres numériques par l'approche empirique.

+ évaluation précise de l'efficacité
+ pas nécessaire d'évaluer empiriquement des exemplaires de très grande taille
− il faut quand même résoudre un bon nombre d'exemplaires

3.2 Analyse empirique et hybride

Toutes les approches décrites dans cette section ont comme point de départ la collecte de couples (x, y) où x est la taille d'un exemplaire et y la quantité de ressource consommée pour cet exemplaire.

Test de puissance : Lorsqu'on ne sait à peu près rien de la consommation de ressources de notre algorithme, ce test peut nous aider à l'exprimer en fonction de la taille des exemplaires. On place les couples relevés sur un graphique avec une échelle log-log. Si on peut y faire passer une droite $\log_a y = m \cdot \log_a x + b$ c'est donc que $y = a^b \cdot x^m$ et nous en déduisons que cette consommation croît de manière polynomiale avec un exposant proche de m et une constante multiplicative évaluée à a^b. Si au contraire la croissance est plus accentuée que linéaire, la consommation est probablement super-polynomiale. Si encore elle tend vers une constante, la consommation est probablement sub-linéaire.

Test du rapport : Lorsqu'on a une certaine idée de la consommation, par exemple qu'elle croît comme la fonction f, cet autre test peut nous aider à vérifier et même à préciser cette hypothèse. On place les couples $(x, y/f(x))$ sur un graphique d'échelle normale. Si une courbe passant par ces points converge vers une constante $b > 0$, notre hypothèse est sensée et b représente une constante multiplicative précisant f. S'il n'y a pas convergence, notre hypothèse s'avère une sous-estimation ; s'il y a convergence vers 0, notre hypothèse s'avère au contraire une sur-estimation. Ce test s'apparente à la règle de la limite.

Test des constantes : Prenant pour acquis la connaissance de la forme générale de la fonction de croissance de la consommation de ressource, disons f, on place les couples $(f(x), y)$ sur un graphique. Si on peut y faire passer une droite, sa pente correspondra à la constante multiplicative et son ordonnée à l'origine, à un certain coût fixe.

Notez que les deux dernières approches sont très proches mais la dernière permet de traiter une fonction à plusieurs paramètres (par exemple si nos exemplaires sont des graphes et qu'on exprime leur taille au moyen du nombre de sommets et d'arêtes) avec un graphique en deux dimensions.

3.3 Analyse théorique

Nous avons maintenant une notation asymptotique afin d´exprimer de façon simple la consommation de ressources d'un algorithme mais nous devons dans un premier temps analyser cet algorithme afin de pouvoir exprimer sa consommation de ressources, typiquement le temps, comme une fonction mathématique de la taille des exemplaires, n. Cette consommation peut varier énormément parmi les exemplaires de même taille. On parlera alors d'analyse...

en pire cas : cherche une fonction pour un pire exemplaire de chaque taille. Elle prédit donc le pire cas pour l'ensemble des exemplaires. Ceci peut par exemple garantir le temps de réponse d'une application critique.

en moyenne : cherche une fonction pour le comportement moyen des exemplaires. Si on utilisera l'algorithme pour un grand nombre d'exemplaires différents, le temps d'exécution moyen nous intéresse. L'analyse en moyenne est souvent plus difficile que l'analyse en pire cas. De plus, pour être vraiment utile, il faudrait connaître la distribution des exemplaires à résoudre, ce qui n'est habituellement pas réaliste. Nous verrons plus tard (et avons déjà vu au chapitre 1) qu'on n'a parfois pas besoin de connaître cette distribution.

Pour un problème donné, soit E_n l'ensemble des exemplaires de taille n et soit $R(e)$ la consommation de ressources de notre algorithme pour l'exemplaire e. La *consommation en pire cas* de notre algorithme est donnée par

$$R_{\text{pire cas}}(n) = \max_{e \in E_n} R(e).$$

Soit $f(e)$ la fréquence d'occurrence de l'exemplaire e. La *consommation en moyenne* de notre algorithme est donnée par

$$R_{\text{moyenne}}(n) = \sum_{e \in E_n} f(e) \cdot R(e) \;/\; \sum_{e \in E_n} f(e).$$

$f(e)$ est habituellement difficile à déterminer et on a souvent recours à des hypothèses simplificatrices (comme par exemple une distribution uniforme, $f(e) = 1$).

3.3.1 Constructions de base d'un langage algorithmique

Dans cette section, nous considérons une ressource en particulier : le temps de calcul.

• OPÉRATIONS ÉLÉMENTAIRES

Une opération est dite *élémentaire* si son temps d'exécution peut être borné supérieurement par une constante qui ne dépend pas de l'exemplaire à résoudre (mais probablement de l'implantation et de l'ordinateur).

Exemple 1 *Les opérations arithmétiques ($+$, $-$, $\times$, $\div$, mod, log, ...) et booléennes ($\wedge$, $\vee$, $\neg$), les comparaisons ($>$, $\geq$, $=$), les affectations ($\leftarrow$) sont des opérations élémentaires.*

Son temps d'exécution est donc dans $\Theta(1)$. On dira qu'une opération élémentaire est effectuée à coût unitaire. **Attention !** Dans certains cas, ces opérations coûteront plus cher : par exemple, l'addition de très grands entiers dont la taille est liée à n.

• SÉQUENCES

Soient P_1 et P_2 deux parties d'un algorithme s'exécutant dans des temps $T_1(n)$ et $T_2(n)$ respectivement. L'exécution de P_1 puis P_2 requiert donc un temps $T_1(n) + T_2(n)$. Par la règle du maximum au chapitre précédent, ce temps est dans $\Theta(\max(T_1(n), T_2(n)))$.

• BOUCLES

nombre d'itérations explicite ("`for`") :

On connaît d'avance le nombre de fois que le corps de la boucle sera exécuté, disons $M(n)$. Soit $T(i,n)$ le temps pris par la $i^{\text{ème}}$ itération du corps de la boucle et c le temps pris pour le contrôle de la boucle. Le temps total pris par la boucle est

$$\sum_{i=1}^{M(n)} (c + T(i,n)) + c.$$

Souvent c est négligeable par rapport à $T(i,n)$ et nous pouvons simplifier à

$$\sum_{i=1}^{M(n)} T(i,n)$$

et même à

$$M(n) \times T(n)$$

lorsque le temps pris par le corps de la boucle est indépendant de l'itération.

Attention ! Ces simplifications sont dangereuses si $M(n) = 0$: le temps total n'est pas nul mais bien c.

nombre d'itérations implicite ("`while`", "`repeat..until`") :

Ici la difficulté est qu'on ne connaît pas d'avance le nombre de fois que le corps de la boucle sera exécuté. On procède généralement en trouvant une fonction des variables impliquées dont la valeur décroît à chaque tour de boucle et dont on connaît le minimum : on obtient ainsi une borne supérieure sur le nombre d'itérations.

Exemple 2 *Fouille dichotomique :*

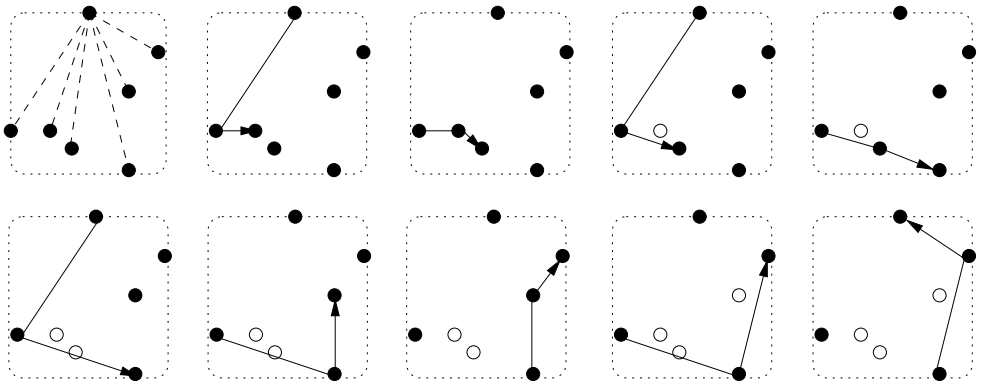
fonction *dicho*($A[1..n]$ *:tableau,* x *:entier) :entier*
  $i \leftarrow 1$; $j \leftarrow n$;
  tant que $i < j$ **faire**
    $k \leftarrow (i + j) \div 2$;
    cas
      $x < A[k]$ *:* $j \leftarrow k - 1$;
      $x = A[k]$ *:* $i \leftarrow k$; $j \leftarrow k$;
      $x > A[k]$ *:* $i \leftarrow k + 1$;
  retourner i;

Ici une fonction appropriée est le nombre d'éléments dans le (sous-)tableau à traiter, $j - i + 1$*, qu'on dénotera* t*. En examinant les trois cas dans le corps de la boucle, on constate que* t *devient soit* 1 *(¨*$x = A[k]$*¨), soit réduit au moins de moitié. Donc à l'itération* ℓ*,* $t \leq n/2^\ell$ *mais l'algorithme s'arrête dès que* $t \leq 1$ *(¨*$i < j$ *¨), ce qui est certainement le cas au plus tard à l'itération* $\lceil \log_2 n \rceil$*. Par conséquent, au plus* $\lceil \log_2 n \rceil$ *itérations seront exécutées.*

Essayez d'appliquer une analyse semblable pour l'algorithme d'Euclide au chapitre 1.

Exemple 3 *L'*enveloppe convexe *d'un ensemble de points est le plus petit ensemble convexe les contenant. Lorsque les points se retrouvent dans un même plan (donc en deux dimensions), on peut calculer leur enveloppe convexe en temps* $\Theta(n \lg n)$ *où* n *est le nombre de points. Voici l'ébauche d'un des premiers algorithmes exhibant cette performance (il faudrait normalement être plus précis) :*

fonction *Graham_ scan(P :ensemble {points}) :liste {enveloppe convexe}*
  $p \leftarrow$ *le point d'ordonnée maximum dans* P *;*
  $L \leftarrow P$ *ordonné autour de* p *en sens antihoraire, en commençant et terminant par* p *;*
  $i \leftarrow 1$ *;*
  tant que $L[i+1] \neq p$ **faire**
    si $L[i], L[i+1], L[i+2]$ *est un virage à gauche*
      alors $i \leftarrow i+1$ *;*
      sinon
        retirer $L[i+1]$ *;*
        $i \leftarrow i-1$ *;*
  retourner*(L)*



L'ordonnancement de P *prend un temps dans* $\Theta(n \lg n)$ *; examinons la boucle. À première vue, cette boucle fait un travail qui est plus que linéaire et peut-être même plus que* $n \lg n$ *puisqu'on recule à l'occasion et qu'on repasse donc plusieurs fois par le même point. Cependant, l'observation clef est qu'à chaque fois qu'on recule, on élimine un point : on ne pourrait donc certainement pas reculer plus de* n *fois, d'où la performance linéaire de cette boucle. Dans l'ensemble, la consommation en temps de calcul* $\in \Theta(n \lg n)$*.*

• CONDITIONNELS

si test
  alors ceci
  sinon cela

La présence d'un choix peut être problématique. Si ¨ceci¨ et ¨cela¨ consomment la même quantité de ressources (à une constante multiplicative près), tout va bien puisque cette consommation est donc une invariante. Mais si ¨ceci¨ implique une consommation d'un autre ordre de grandeur que celle de ¨cela¨, que fait-on ?

Exemple 4

pour $k = 1$ **à** n **faire**
  si $i < j$
    alors /* *exécuté* $\Theta(n)$ *fois* */
      $i \leftarrow i+1$;
    sinon /* *exécuté* $\Theta(\lg n)$ *fois* */

trier$(A[1..n])$;
$j \leftarrow A[i]$;

On peut considérer le scénario pessimiste et prendre le maximum des deux, ou encore faire une analyse plus fine en s'inspirant de l'analyse amortie (section **??**).

• RÉCURSIVITÉ

Exemple 5 *Fouille dichotomique :*

$$T(n) = \begin{cases} c_1 & \text{si } n = 1;\ c_1 > 0 \\ T(n/2) + c_2 & \text{sinon};\ c_2 > 0,\ n \text{ puissance de 2} \end{cases}$$

Rappel (LOG2810) [facultatif]

Une *relation de récurrence* ramène le calcul des valeurs d'une suite à celui des valeurs précédentes. Il existe une technique dite de l'*équation caractéristique* qui permet de résoudre certaines classes de récurrences presque automatiquement.

Récurrence linéaire homogène à coefficients constants, de la forme

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0,$$

où les t_i sont les valeurs que l'on cherche (la suite mathématique (t_n)) et les a_i sont des constantes.

Soient $r_1, \ldots, r_\ell$ les zéros (distincts) de multiplicité $m_1, \ldots, m_\ell$ respectivement de l'*équation caractéristique*,

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0,$$

correspondant à cette récurrence. La solution générale est alors de la forme

$$t_n = \Sigma_{i=1}^{\ell} \Sigma_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$$

Récurrence non homogène de la forme

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n),$$

où b est une constante et $p(n)$ un polynôme en n de degré d.

Son polynôme caractéristique est

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1}.$$

Note : Une combinaison linéaire générale de solutions n'est plus nécessairement une solution : il faut la réinjecter dans la récurrence, ce qui précise certaines constantes.

Exemple 6

$$t_n = \begin{cases} 1 & \text{si } n = 0 \\ 4t_{n-1} - 2^n & \text{sinon} \end{cases}$$

polynôme caractéristique : $(x - 4)(x - 2)$

$$t_n = c_1 4^n + c_2 2^n \text{ dans l'ordre exact de } 4^n \text{ ?}$$

. . . où $c_1 4^n + c_2 2^n = 4(c_1 4^{n-1} + c_2 2^{n-1}) - 2^n \Rightarrow c_2 2^n = (2c_2 - 1) \cdot 2^n \Rightarrow c_2 = 1$
Pour déterminer c_1 : $1 = t_0 = c_1 + 1 \Rightarrow c_1 = 0$.

$$t_n = 2^n \text{ et donc } t_n \in \Theta(2^n)$$

Généralisation supplémentaire : De la même façon, une récurrence de la forme

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \cdots ,$$

où les b_i sont des constantes distinctes et les $p_i(n)$ des polynômes en n de degré d_i respectivement ont pour polynôme caractéristique :

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \cdots .$$

Changement de variable :

Exemple 7

$$T(n) = 4T(n/2) + n^2 \text{ , } \textit{où } n \textit{ est une puissance de 2, } n \geq 2$$

Remplaçons n par 2^i

$$T(2^i) = 4T(2^{i-1}) + (2^i)^2$$

et définissons une nouvelle suite $t_i = T(2^i)$

$$t_i = 4t_{i-1} + 4^i \text{ , } \textit{d'une forme qu'on sait résoudre}$$

poly. caract. : $(x-4)^2$

$$t_i = c_1 4^i + c_2 i 4^i$$
$$T(n) = c_1 n^2 + c_2 n^2 \lg n$$

où

$$\begin{aligned} n^2 &= c_1 n^2 + c_2 n^2 \lg n - 4(c_1 n^2/4 + c_2 (n^2/4)(\lg n - 1)) \\ &= c_2 n^2 \end{aligned}$$

Donc $c_2 = 1$ et

$$T(n) \in \Theta(n^2 \lg n \mid n \textit{ puissance de } 2)$$

peu importe les conditions initiales.

Fin du rappel facultatif

Voici une récurrence très importante pour l'analyse des algorithmes (diviser-pour-régner) :

Soient les constantes $n_0, \ell \geq 1, b > 1, k \geq 0$ et $c > 0$.

Soit $T : \mathbb{N} \to \mathbb{R}_+$ telle que

$$T(n) = \ell\, T(n/b) + cn^k, \qquad n > n_0$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \lg n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases} \qquad (\star)$$

3.3.2 Opération baromètre

Une opération *élémentaire* peut être utilisée comme *baromètre* si elle est exécutée au moins aussi souvent que n'importe quelle autre. Cela simplifie considérablement l'analyse d'un algorithme puisqu'il prendra un temps dans l'ordre exact du nombre de fois qu'on exécute le baromètre.

Exemple 8 *Soit l'¨algorithme¨ suivant :*

$k \leftarrow 0$;

pour $i = 1$ **à** n **faire**

$k \leftarrow 4 \times k + 5$;
pour $j = i$ **à** n **faire**
  si $k > j$
    alors $k \leftarrow k - i$;
    sinon $k \leftarrow k + i$;

*Son analyse est simplifiée si on prend comme baromètre le test du conditionnel (¨***si** $k > j$ *¨), ce qui est correct en autant que* $n \geq 1$*. On aura alors :*

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} c = \sum_{i=1}^{n} (n - i + 1) \cdot c = c \cdot (n^2 - \frac{n(n+1)}{2} + n) = c \cdot (\frac{n^2}{2} + \frac{n}{2}) \in \Theta(n^2)$$

On peut aussi parfois choisir un baromètre de plus haut niveau, comme traiter une arête d'un graphe ou comparer deux éléments d'un tableau.

3.3.3 Analyse amortie

Parfois plusieurs exécutions d'une même opération ont des consommations de ressources très variables. Par exemple : l'allocation de mémoire prend parfois plus de temps à cause de l'intervention du glaneur de cellules ; certaines structures de données, comme les arbres équilibrés, nécessitent plus ou moins de travail lors de insertion ou du retrait d'un élément en fonction de l'état courant de la structure. D'autres fois, il s'agit d'opérations différentes pour une même structure de données mais qui ont des consommations très différentes. La consommation totale est souvent bien inférieure au produit du nombre d'opérations exécutées et de la consommation la plus coûteuse (raisonnement en pire cas pour une opération individuelle). Nous verrons ici comment pratiquer une analyse plus fine, l'analyse amortie. La *consommation amortie* d'une seule opération parmi une séquence d'opérations est calculée comme la consommation en pire cas de l'ensemble de cette séquence divisée par le nombre d'opérations. Là où elle s'applique, c'est une façon d'effectuer une espèce d'analyse "en moyenne" sans avoir recours à la probabilité d'occurrence de chaque type d'opération.

Pour un problème donné, soient m opérations dans la séquence d'intérêt. La *consommation amortie* d'une opération individuelle pour un exemplaire de taille n est donnée par

$$R_{\text{amortie}}(n) = \frac{R_{\text{pire cas}}(n)}{m}.$$

Le truc de comptabilité. On doit d'abord deviner le coût amorti de l'opération qui nous intéresse (ou au moins une sur-estimation de celui-ci), disons γ, ce qui demande un peu d'intuition ou d'essai/erreur. On se crée ensuite un compte en banque fictif qui ne doit jamais être à découvert et qui contient initialement 0$. Chaque fois qu'on exécute l'opération, on met $\gamma$$ en banque. On retirera par contre 1$ pour chaque unité de ressource consommée cette fois-ci par l'opération. Si on peut démontrer que le compte ne sera jamais à découvert alors m exécutions de l'opération auront une consommation d'au plus $m\gamma$ quelque soit m :

$$R_{\text{amortie}}(n) = \frac{R_{\text{pire cas}}(n)}{m} \leq \frac{m\gamma}{m} = \gamma$$

donc $R_{\text{amortie}}(n) \in \mathcal{O}(\gamma)$.

Exemple 9 *compteur binaire*

procédure *incrémenter(*$A[1..n]$ *:tableau)*

$j \leftarrow n+1$;
répéter
$j \leftarrow j-1$;
$A[j] \leftarrow 1 - A[j]$;
jusqu'à ce que $A[j] = 1$ **ou** $j = 1$

pour $i = 1$ **à** m **faire**
incrémenter(A) *;*

Au pire, un appel à *incrémenter* nécessitera n tours de boucle (lorsque A ne contient que des ¨1¨). Une exécution de cette procédure prend donc un temps dans $\Theta(n)$ en pire cas. On en conclut que m exécutions consécutives à partir d'un tableau A ne contenant que des ¨0¨ prennent un temps dans $\mathcal{O}(mn)$; mais prennent-elles un temps dans $\Theta(mn)$? Appliquons l'analyse amortie.

Devinons un $\gamma = 2$. Chaque appel à *incrémenter* dépose 2$ en banque et chaque tour de boucle lors de cet appel retire 1$. Remarquez que le dernier tour de boucle exécuté lors d'un appel (sauf au débordement du compteur) change un ¨0¨ en ¨1¨ alors que tous les tours précédents changent un ¨1¨ en ¨0¨ : par conséquent, un appel de k tours de boucle décroît le nombre de ¨1¨ dans le compteur de $(k-1)-(1) = k-2$ et fait diminuer le compte en banque de $(k-2)$$ également.

Initialement, on a le même nombre de $ en banque que de ¨1¨ dans le compteur (aucun). En tout temps, on aura donc autant de $ en banque que de ¨1¨ dans le compteur. Puisque le nombre de ¨1¨ n'est jamais négatif, notre compte ne sera jamais à découvert. Ainsi m exécutions consécutives de *incrémenter* à partir d'un tableau A ne contenant que des ¨0¨ prennent un temps dans $\mathcal{O}(2m) = \mathcal{O}(m)$. (On en déduit facilement que ce temps est aussi dans $\Theta(m)$.) Donc $R_{\text{amortie}}(n) \in \Theta(1)$.

3.3.4 Analyse approximative

Il est possible que nous n'arrivions pas à formuler de façon précise la consommation de ressoures d'un algorithme ($T(n) = f(n)$) mais tout au plus à la borner ($T(n) \leq f(n)$) : par exemple, si une analyse pessimiste de conditionnels ne peut être évitée ou suite à une analyse amortie.

Dans ce cas, nous ne pouvons utiliser la notation Θ et devons nous rabattre sur $\mathcal{O}$ (ou Ω).