Notions d'algorithmique

Table des matières

1 Introduction

Un algorithme est une séquence d'étapes de calcul bien définie permettant de résoudre un problème, prenant en entrée un ensemble de valeurs (paramètres) et donnant en sortie un autre ensemble de valeurs. Le problème va spécifier la relation attendue entre les valeurs d'entrées et celles de sorties, et l'algorithme décrit une procédure précise qui permet d'obtenir cette relation.

Exemple 1.1 *Algorithme du pgcd, crible d'Eratosthene, recette de cuisine.*

1.1 Structure de contrôle

1. les structures de contrôle : une structure de contrôle est une commande qui contrôle l'ordre dans lequel les di□érentes instructions d'un algorithme ou d'un programme informatique sont exécutées.
   (a) séquences

(b) conditionnelles

(c) boucles

1.2 Structures de données

1. les structures de données : une structure de données est un moyen de stocker et d'organiser des données, pour faciliter l'accès à ces données et leur modification.
   (a) constantes
   (b) variables
   (c) tableaux
   (d) structures récursives (listes, arbres, etc.)

1.3 Ce qu'on attend d'un algorithme

Un algorithme doit, autant que possible, répondre à un problème

Definition 1.2 *Un algorithme est dit correct si pour chaque entrée qu'on peut lui fournir il produit en sortie celle qui est attendue : il résoud le probleme donné.*

Remark 1.3 *Un algo incorrect peut tout a fait avoir son utilité, pour peu que le taux d'erreur soit contrôlable (exemple : test de primalité probabilistique).*

Definition 1.4 *Un algorithme termine lorsque pour toutes les instances qu'on peut lui donner en paramètres, il prend un temps fini pour donner les valeurs de sortie.*

2 Conception des algorithmes

1. Approche *diviser pour régner*
2. Algorithme *glouton*
3. Backtracking (déjà vu ?)
4. Recherche exhaustive
5. Approche heuristique
6. Approche aléatoire

2.1 Approche incrémentale

Un exemple d'algorithme utilisant une approche incrémentale est le tri par insertion. On a un jeu de cartes dans la main gauche. A chaque étape, on met à la place voulue dans la main droite (qui était libre au début) une carte du jeu de la main gauche : on fait grandir le tas de cartes de la main droite.

2.2 Recherche exhaustive

On charche la solution à un problème et on teste toutes les solutions possibles (potentiellement un nombre infini) Par exemple l'attaque par force brute : on cherche le mot de passe de quelqu'un, et on teste toutes les possibilités les unes après les autres. Cela ne marchera que si le mot de passe est court, et c'est évidemment exponentiel en la taille du mot de passe.

On peut compléter une recherche exhaustive par une recherche par dictionnaire qui permet de tester certains mots plus probable, par exemple le nom du chien de la personne visée.

Un autre exemple : un rubik's cube mélangé se résoud en appliquant un certain nombre de rotations de certaines de ses faces. Une attaque par force brute va essayer chaque mouvement qui utilise une rotation, puis chaque mouvement qui en prend deux, etc. jusqu'à résoudre le cube. Au delà de dix mouvements, c'est déjà largement trop long (une rotation d'une face se fait en temps constant, mais nécessite un certain nombre d'opérations de base).

2.3 Algorithme glouton

Choix optimum local, en espérant obtenir un optimum global. Dans le cas du rendu de monnaie par exemple, l'algorithme glouton commence par rendre la pièce la plus grande plus petite que ce qqu'il faut rendre, puis est réappelé avec le porte-monnaie réactualisé sur la nouvelle somme à rendre.

2.4 diviser pour régner

On divise un probleme d'une certaine taille en sous-problèmes analogues de taille plus petite. Par exemple, le tri rapide, le tri fusion, la recherche dichotomique. Cette approche fournit des algorithmes plus rapides. Lees algorithmes récursifs utlisent généralement une approche *diviser pour régner*

- **Diviser** le problème en un certain nombre de sous-problèmes
- **Régner** sur les sous-problèmes en les résolvant de manière récursive. Le cas échéant, on les résoud directement, ou en utilisant un autre algorithme mieux adapté (exemple, le tri par insertion qui est e□cace sur un petit nombre d'éléments, et qu'on peut utiliser dans un tri rapide)
- **Recombiner** les solutions obtenues des sous-problèmes pour en déduire la solution du problème de départ.

Autre exemple, le tri rapide, le tri fusion, l'exponentiation rapide, recherche dans un dictionnaire

2.5 Heuristique

Une heuristique est un algorithme qui fournit rapidement une solution pas nécessairement optimale à un problème d'optimisation. Ce n'est donc pas un algorithme exact. Sur certains problèmes, un algorithme qui donnera une solution exacte pourra être de de complexité exponentielle (la complexité est expliquée plus loin), donc très longs.

2.6 Et des mélanges

On n'est pas obligé de suivre un modèle de conception d'algorithme, il peut être avantageux de prendre le meilleur dans chaque modèle. Par exemple, on sait que sur les petites valeurs, le

tri par insertion est très e□cace, mais que sur les grandes valeurs, le tri rapide l'emporte. On choisit donc d'utiliser le tri rapide, avec pour restriction que, si à un moment donné, l'algorithme `quicksort` est appelé sur moins de 15 éléments, alors on trie ces élements avec le tri par insertion.

Et on a déjà parlé de l'attaque force brute que l'on peut compléter par une attaque par dictionnaire.

3 Preuve d'algorithme

3.1 Algorithme à boucles

On veut d'un algorithme qu'il termine en donnant une réponse exacte au problème qu'il est sensé résoudre.

Pour prouver que c'est le cas, on utilise par exemple, dans le cas où il y a une boucle, un *invariant de boucle* :

Sur le modèle d'un raisonnement par récurrence, on vérifie qu'une propriété est vraie avant la première boucle (Initialisation) et que pour toute boucle qui sera e□ectuée, si elle vraie en début de boucle, elle l'est aussi après l'exécution de la boucle (Conservation), et que le fait qu'elle soit vraie en fin d'exécution de l'algorithme implique que celui-ci termine bien en donnant la réponse attendue (terminaison). L'invariant fournit une propriété utile qui permet de montrer la validité de l'algorithme.

Exemple : calcul de la factorielle avec une boucle `for`.

- On initialise une variable à 1.
- A chaque étape d'une boucle pour , de i = 1 à n (pour l'exemple !), on multiplie la variable par i.
- Invariant de boucle : en fin de la i-eme boucle et au début de la (i + 1)- ème, la variable vaut i!

C'est vrai avant la premier boucle, l'hérédité est facile, et on en déduit que l'algorithme renvoie bien la factorielle.

On peut faire de même avec une boucle while.

3.2 Cas récursif

Dans le cas récursif : il faut être sûr (en général !) que l'algorithme termine (ce dont on n'a pas besoin dans le cas d'une boucle `for`). Il faut donc qu'il y ait des cas de bases sur lesquels l'algorithme termine et donne la réponse souhaitée, et que les autres cas s'y rapportent en un nombre fini d'étapes. Lorsque l'on appelle l'algorithme sur i et qu'il se rappelle sur j, il faut que j soit plus petit que i pour un ordre donné : on a donc besoin d'une relation bien fondée (il existe des éléments minimaux pour tout sous-ensemble, et (il y a équivalence en fait) il n'existe aucune suite strictement décroissante infinie). Avec une telle relation, et avec les cas de bases correspondant aux cas minimaux, on prouve que l'algorithme termine.

Exemple 3.1 *Contre-exemple : syracuse, dont on ne sait pas si elle termine (mais dont on le suppose fortement)*

Exemple : calcul récursif du log d'un élément, montrer pourquoi ça termine.

4 Complexité

Admettons qu'on dispose de deux algorithmes di□érents pour le même problème, il est intéressant de savoir lequel sera le plus avantageux : s'ils sortent la même réponse pour le problème, lequel sera le plus rapide à le faire (on regarde le temps d'exécution), lequel utilisera le moins de mémoire (espace mémoire requis). Il peut y avoir encore d'autres critères (comment sont utilisées les di□érentes ressources de l'ordinateur sur lequel tourne le programme), mais on ne s'intéresse ici qu'à la complexité temporelle et à la complexité spatiale.

4.1 Un peu de maths

Soient f et g deux applications de $\mathcal{N}^*$ dans $\mathcal{N}^*$. On dit que f est un "grand O" de g, ce qu'on note $f = \mathcal{O}(g)$, s'il existe C une constante, t.q. pour tout n, $f(n) \leq Ag(n)$. Il s'agit d'une relation transitive (si $f = \mathcal{O}(g)$ et que $g = \mathcal{O}(h)$ alors $f = \mathcal{O}(h)$). On a aussi $f = \mathcal{O}(g) \Rightarrow \sum_{1\leq k\leq n} f(k) = \mathcal{O}(\sum_{1\leq k\leq n} g(k))$

On note $f = \Theta(g)$, si f est un grand O de g et que g est un grand O de f.

Soient f et g deux applications de $\mathcal{N}^*$ dans $\mathcal{N}^*$. On dit que f est un "petit o" de g, ce qu'on note $f = o(g)$, si $\frac{f(n)}{g(n)} \to 0$. Il s'agit aussi d'une relation transitive.

On peut être amené à résoudre des récurrences pour calculer la complexité d'un algorithme (des exemples seront plus détaillés plus tard) :

Pour résoudre les récurrences Si on a $c(n+1) = ac(n) + f(n)$, il faut étudier la fonction f. Plus précisément, on voit que ce qui précède s'écrit aussi $\frac{c(n+1)}{a^{n+1}} = \frac{c(n)}{a^n} + \frac{f(n)}{a^n}$, ce qui donne $\frac{c(n)}{a^n} = \frac{c(1)}{a} + \sum \frac{f(k)}{a^k}$, et qu'il faut donc connaitre le comportement de la série de terme général $\frac{f(k)}{a^k}$.

- $f(n) = \mathcal{O}(a^n)$, alors $c(n) = \mathcal{O}(na^n)$
- $f(n) = \mathcal{O}(b^n)$, avec $b > a$, alors $c(n) = \mathcal{O}(b^n)$
- La série de terme général $f(n)/a^n$ converge, alors $c(n) = \mathcal{O}(a^n)$

De même, on peut étudier les récurrences de type diviser pour régner : $c(n) = ac(\lfloor n/2 \rfloor) + ac(\lceil n/2 \rceil) + f(n)$, où a et b sont deux constantes qu'on peut supposer non simultanément nulles (souvent $a = b = 1$ ou $a + b = 1$).

On note $\alpha = \lg(a + b)$ et on regarde à nouveau la fonction f, qu'on suppose croissante (quitte à remplacer f par une fonction g croissante la dominant pour n'obtenir qu'une majoration) :

- $f(n) = \Theta(n^\beta)$, avec $\beta < \alpha$ alors $c(n) = \Theta(n^\alpha)$
- $f(n) = \Theta(n^\alpha)$, alors $c(n) = \Theta(n^\alpha \ln n)$
- $f(n) = \Theta(n^\beta)$, avec $\beta > \alpha$ alors $c(n) = \Theta(n^\alpha)$

4.2 Les principales classes de complexité

Un algorithme qui e□ectue $5n^3 + 4n\log(n) + 2$ opérations de bases se comportera comme un algorithme qui e□ectue n^3 opérations de base, dans le sens où si on double n, le temps demandé pour exécuter l'algorithme sera multiplié par 8 dans les deux cas. On va séparer les di□érents cas en quelques classes dont certaines sont évoquées ici.

On peut avoir une complexité *logarithmique*, en $\Theta(\ln n)$, une complexité linéaire (si on veut la longueur d'une liste par exemple), une complexité polynomiale (tri bulle et par insertion : quadratique), exponentielle ...

On peut considérer qu'un algorithme en complexité logarithmique ne coûtera pas tellement plus que s'il était constant : si on borne la taille des données à $n = 10^{15}$, on a $\ln(n) = 35$...

Vous remarquerez que dire d'un algorithme qu'il est au moins en $\mathcal{O}(n)$ n'apporte pas tellement d'informations !

A titre d'exemple, gardez en mémoire que l'age de l'Univers est de $4,32 \times 10^{17}$ secondes (source : Wolfram). Un algorithme qui est de complexité exponentielle 2^n e□ectue 10^{30} opérations pour $n = 100$. Un algorithme en n^2 e□ectue 10^{18} opérations pour $n = 10^9$, et pour la même valeur de n, $\ln(n) < 21$.

4.3 Complexité temporelle

Une fois que l'on sait qu'un programme termine et fait ce qu'on lui demande, on s'intéresse au temps qu'il met à le faire, ou plus exactement à l'ordre de grandeur du nombre d'instructions de base qu'il va lui falloir utiliser, en fonction de la taille du ou des paramètres, ou d'un paramètre donné. On cherche en général à connaître le temps en moyenne et dans le pire cas, qui correspondent à deux utilisations di□érentes : pour certains problèmes, il su□t que celui qui attend la réponse soit satisfait dans un délai raisonnable, mais puisse attendre, de manière exceptionnelle un peu plus longtemps, alors on regardera la complexité en moyenne. Par contre, si on veut bien attendre un certain temps qu'il ne faut pas dépasser, on regardera la complexité dans le pire cas.

Exemples Exemple : `length` en $O(n)$, où n est la taille de la liste, `quicksort` et tri fusion en $O(n \ln(n))$ où n est la taille de la liste,

Tri à bulle : quadratique en moyenne : on a un échange à faire pour chaque inversion, or le nombre moyen d'inversion sur les permutations de taille n est $\frac{n(n-1)}{4}$, en moyenne, la complexité est donc quadratique. Dans le pire cas, la liste est triée à l'envers, on a $n(n-1)/2$ inversions, cas quadratique aussi.

En pratique Si on a un algorithme cubique avec un petit coe□cient et au autre algorithme, quadratique, avec un coe□cient énorme, on pourra choisir l'algorithme cubique. De même, si un algorithme est en moyenne comme dans le pire cas en $O(n \log(n))$(cf tri par tas), mais qu'un autre a un pire cas en $O(n^2)$ et un cas moyen en $O(n \log(n))$(cf quicksort), mais qu'en pratique le deuxième est deux fois plus rapide que le premier, on choisira le deuxième algorithme...

4.4 Complexité spatiale

On s'intéresse parfois aussi à la complexité en espace, c'est à dire à la taille de la mémoire que l'on va utiliser.

Le tri fusion (une implémentation du) sur un vecteur/tableau va prendre un tableau de n éléments, et les recopier dans deux vecteurs de taille $n/2$: il utilise donc en mémoire un espace supplémentaire en $\mathcal{O}(n)$.

5 Analyse d'algorithme sur des exemples

L'analyse d'algorithme consiste dans nombre de cas à prévoir les ressources qui seront nécessaires à cet algorithme (mémoire, temps de calcul, etc.). C'est donc l'application de ce qui précède.

On commence par définir la $taille\ de\ l'entrée$, que l'on fera varier, et en fonction de laquelle sera calculé le temps mis par l'algorithme. Ici, c'est le nombre d'éléments à trier, ce qui correspond à la notion assez naturelle du nombre d'éléments constituant l'entrée. On aurait pu choisir le nombre de bits nécessaires pour coder les éléments d'entrée (dans le cas d'une méthode codant la multiplication de deux entiers).

Le $temps\ d'exécution$ est le nombre d'opérations élémentaires que l'algorithme e□ectue. En général, ces opérations élémentaires sont : les opérations arithmétiques, l'accès à l'élément de tête d'une liste, l'accès à n'importe quel case d'un vecteur (==tableau, ils ne connaissent pas tous les vecteurs), l'a□ectation d'une valeur à une case d'un tableau, etc. Il faut garder en tête qu'il s'agit d'une approximation et que cela dépend en fait du modèle de calcul utilisé par la machine : si ce modèle dispose d'un test de primalité en temps constant (c'est un exemple tout a fait théorique, et assez irréaliste...), alors chaque appel à ce test se fera en temps constant.

5.1 Les tours de Hanoï

On a n rondelles à placer, trois piquets, une rondelle ne peut être que sur une autre rondelle plus grande qu'elle, et on ne déplace qu'une rondelle à la fois, pour avoir, à la fin, toutes les rondelles rangées sur le dernier piquet. Dans la configuration initiale, elles sont toutes rangées sur le premier piquet.

On veut connaître $c(n)$ le nombre de mouvements pour ranger les n rondelles. On peut déplacer les $(n-1)$ rondelles du dessus vers le deuxième piquet, déplacer la plus grande rondelle sur le troisième piquet, puis redéplacer les $(n-1)$ rondelles du deuxième piquet vers le troisième. Si on fait ça, $c(n) = 2c(n-1) + 1$. On a ici une récurrence linéaire.$Ex.$: Trouver la complexité de cet algorithme.

5.2 Analyse du tri par insertion

A propos du tri par insertion sur un vecteur/tableau (en place, donc pas d'étude de la complexité spatiale) :

le tri insertion comporte une boucle `for` de 2 à n. A chaque étape de cette boucle, on a une a□ectation (l'élément à insérer j est mis dans une variable), puis on recherche sa place, en décalant à chaque fois d'une case les éléments plus grands, jusqu'à le placer. S'il y a k éléments plus grands que j déjà placés (k peut varier de 0 à $j-1$), on aura fait k réa□ectations, $k+1$ tests et deux a□ectations pour j.

On a donc un nombre d'opérations élémentaires égal, dans le pire des cas, à $\sum_{j=2}^{n}(j-1) + j + 2 = \mathcal{O}(n^2)$, dans le meilleur des cas, à $\mathcal{O}(n)$.

On peut aussi regarder la complexité $M(n)$ en moyenne. Or, pour chaque élément j que l'on rerange, on a un nombre de test et de réa□ectation linéaire en le nombre d'éléments plus grand

que j et placés avant (il s'agit des inversions). Dans le cas moyen, une permutation contient $n(n-1)/4$ inversions (*Ex. :* Le prouver (regarder une permutation et celle qu'on obtient en changeant i en $n-i$, puis faire la moyenne des inversions sur toutes les permutations et tous leurs "inverses")), on a donc encore une complexité quadratique.

5.3 Analyse du tri fusion

On veut connaitre $T(n)$ le pire temps d'exécution. On considère une liste d'éléments, sur laquelle on appelle une procédure `diviser` qui crée deux sous-séquences de taille $n/2$ en temps linéaire. On trie récursivement ces deux sous-séquences, en temps au plus $T(n/2)$ pour chacun, puis on les fusionne (temps linéaire).

On a donc $T(n) = 2T(n/2) + \Theta(n)$ (avec, bien sûr, $T(1)$ constant), soit $T(n) = \Theta(n) + 2\Theta(n/2) + 4\Theta(n/4) + \ldots \log n\Theta(1) = \Theta(n \log n)$

5.4 Complexité minimale

Il peut être intéressant aussi de connaître la complexité minimale : si un problème nécessite au moins un temps quadratique, qu'on ne cherche pas à créer un algorithme en temps linéaire. On va voir ici que les tris par comparaison, qui ne nécessitent aucune hypothèse préalable sur les éléments que l'on veut trier (on ne connait pas leur distribution, ni la valeur du nombre maximal, etc.), nécessitent au moins un temps en $n \ln(n)$.

Le problème du tri sur n éléments peut se voir comme la recherche d'une permutation de S_n qui, appliquée à l'entrée, renvoie une liste/vecteur trié(e) des éléments. On peut modéliser l'algorithme par un arbre binaire qui e□ectue des comparaisons à chaque noeud (en fonction du résultat de la comparaison, on visite le sous-arbre gauche ou le sous-arbre droit), et dont les feuilles représentent des permutations. Puisque le cardinal de S_n est $n!$, il faut qu'il y ait au moins $n!$ feuilles. Le temps de l'algorithme sur une entrée sera la profondeur de la feuille qui est la solution attendue. Minimiser la complexité en moyenne de l'algorithme revient à minimiser la profondeur moyenne des feuilles de l'arbre. Quelque soit le nombre de feuille, c'est l'arbre complet qui permet de minimiser cette moyenne (preuve par l'absurde : sinon, on a deux feuilles dont la profondeur di□ère d'au moins deux et on peut construire un arbre qui donne une meilleure moyenne). De même, si on veut que le pire cas soit minimisé, il faut choisir l'arbre complet.

Un tel arbre, possédant $n!$ feuilles, sera de hauteur h, où $2^{h-1} < n! \leq 2^h$. La formule de Stirling ($n! \sim \sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n$) donne $h \geq n \cdot \log_2(n) - n/\ln 2 = \Theta(n \log n)$, ce dont on déduit qu'en moyenne, comme dans le pire cas, aucun algorithme ne fera mieux que $\Theta n \ln n$.