Motivation: À un problème donné, il existe parfois plusieurs façons de trouver une solution. On souhaite donc pouvoir comparer ces méthodes.

I Introduction

Def 1: Un *algorithme* est une suite finie d'étapes décrivant un traitement. Chaque étape est formée d'un nombre fini d'opérations dont chacune est:
- → définie de façon rigoureuse et non-ambiguë
- → effective (pouvant être réalisée sur une machine)

exemple 2: A:= "prendre x comme entrée; multiplier x par 5; ajouter 2 au résultat obtenu; renvoyer ce deuxième résultat" est un algorithme qui calcul $5x+2$.

Def 3: Parmi les opérations possibles, on en choisit certaines, *les opérations fondamentales*, et on leur associe un coût.

Exemple 4: Si on choisit la multiplication d'entiers et l'addition d'entiers comme opérations fondamentales avec des coûts respectivement égaux à c_x et c_+; alors l'algorithme A a un coût constant de $c_x + c_+$.

Remarque 5: De manière général nous nous interesserons au nombre d'opération fondamental.

Def 6: Une *instance* est un ensemble de données d'entrée d'un algorithme. A chaque instance on associe *sa taille* qui est la place qu'elle prend en mémoire.

exemple 7:
- Sur une architecture 32 bit, tout les nombres de 0 à $2^{32}-1$ ont la même taille.
- Une liste de n éléments est de taille n.

Def 8: Soit x une instance, on note $C(x)$ le nombre d'opérations fondamentales effectuées lors de l'éxecution d'un algo A donné sur l'entrée x. On peut donc définir les complexité au pire cas et en moyenne d'un algorithme A:

→ pire cas $\quad C_{max}(n) = \max\{C(x) \,/\, x \text{ de taille } n\}$.

→ moyenne $\quad C_{moy}(n) = \sum_{\substack{x \text{ de} \\ \text{taille } n}} p(x)\, c(x)$ où $p(x)$ est la probabilité d'avoir l'instance x sur toutes les instances de taille n.

Exemple 9: On dispose d'un tableau de n chiffres (de 0 à 9) et on dispose de l'algorithme suivant qui dit si oui ou non il y a un 0 dans le tableau.

```
ExisteO(T)
    Pour i=0 à T.taille-1
        si T.(i)=0
        alors retourner "oui"
    retourner "Non"
```

Si on considère la comparaison à 0 comme une opération fondamentale on obtient les complexités suivantes:

$$C_{max}(n) = n \qquad C_{moy}(n) = \sum_{i=1}^{n} \left(\frac{9}{10}\right)^{i-1} \frac{i}{10} + \left(\frac{9}{10}\right)^{n} n$$

Remarque 10: Ici on a pris une probabilité uniforme sur nos tableaux. Ce n'est pas toujours le cas, par exemple lorsqu'on étudie des algorithmes sur des mots dans une langue particulière.

Def 11 : On note $C^s(x)$ l'espace mémoire utilisé lors de l'exécution de l'algorithme A avec x comme entrée. Par analogie aux complexités temporelles on définit les complexités spatiales :

- au pire $C^s_{max}(n) = \max \{ C^s(x) / x \text{ de taille } n \}$
- moyenne $C^s_{moy}(n) = \sum_{\substack{x \text{ de} \\ \text{taille } n}} p(x) C^s(x)$

Nous nous intéressons désormais à des majorations ou des ordres de grandeur asymptotiques des complexités donc nous introduisons les notations de Landeau :

Notation 12 : On note

$f(n) = O(g(n))$ si $\exists c>0, \exists N \geq 0 \quad \forall n \geq N \quad f(n) \leq c g(n)$

$f(n) = \Omega(g(n))$ si $g(n) = O(f(n))$

$f(n) = \Theta(g(n))$ si $g(n) = O(f(n))$ et $f(n) = O(g(n))$

Exemple 13 :

si $f(n) = n^3 + n$ alors $f(n) = O(n^3)$

si $f(n) = n \log n + 5n$ alors $f(n) = O(n \log n)$

II Méthodes de calcul de complexité

Si certains algorithmes demande un calcul direct de la complexité (ex : tri polyphasé) ; d'autres peuvent être approchés pas des méthodes plus particulières, notamment les algorithmes récursifs.

a) Cas général des algorithmes récursifs.

Cette méthode est idéale pour les algorithmes suivant le paradigme "diviser pour régner", par exemple le tri fusion qui a une complexité au pire cas qui suit la relation de récurrence $C(n) = 2C(\lfloor \frac{n}{2} \rfloor) + n$

Théorème 14 : Soit $T(n) = aT(\lfloor \frac{n}{b} \rfloor) + f(n)$ $a \geq 1$ $b > 1$

- si $f(n) = O(n^{\log_b a - \varepsilon})$ pour $\varepsilon > 0$ alors $T(n) = \Theta(n^{\log_b a})$
- si $f(n) = \Theta(n^{\log_b a})$ alors $T(n) = \Theta(n^{\log_b a} \log n)$
- si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour $\varepsilon > 0$ et $\forall n \geq N \quad a f(\lfloor \frac{n}{b} \rfloor) \leq c f(n)$ pour $c, n > 0$ } alors $T(n) = \Theta(f(n))$

Exemple 15 : puisque $n = \Theta(n^{\log_2 2})$ la complexité du tri fusion est $\Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$ dans le pire des cas.

b) Séries génératrices

Lorsque le coût suit une relation de récurrence (E) qui s'y prête, il est possible d'utiliser les séries génératrices

Prop 16 : Méthode

i) Considérer $f(x) = \sum_{n \geq 0} C(n) x^n$

ii) Utiliser (E) pour développer la somme et en déduire une relation sur f.

iii) Expliciter $f(x)$ en fonction de x.

iv) Développer f en série entière.

v) En déduire $C(n)$.

III Complexité amortie

Dans certains cadres, la complexité moyenne ou pire n'est pas adaptée. Par exemple lorsqu'on manipule des tableaux dynamiques.

Exemple 17 : Dans un tableau dynamique l'ajout d'un élément peut rencontrer deux situations :

- soit le tableau n'est pas plein dans ce cas on ajoute simplement l'élément.
- soit le tableau est plein dans ce cas il faut réallouer de la mémoire ailleurs et recopier le tableau précédent.

La complexité en moyenne n'est pas adaptée car on SAIT qu'après un agent couteux, on a le temps avant de retomber dessus

1) Méthode de l'agrégation

Prop 18 Si une suite de n opérations coûte $T(n)$ alors une opération coûte $\frac{T(n)}{n}$.

Exemple 19 Compteur binaire. Supposons avoir une variable que l'on incrémente en partant de 0. Naïvement le coup d'une incrémentation de n à n+1 est en $O(\log n)$. Or incrémenter de 0 à n coûte $T(n) = \sum_{i=0}^{\log n} \lfloor \frac{n}{2^i} \rfloor \leq 2n$ donc le coût amortie par agrégation d'une incrémentation est inférieure à **2**.

2) Méthode comptable.

Le principe de cette méthode est de surévaluer certaines opérations qui se font nécessairement avant d'autres.

Exemple 20: Naturellement les coûts des opérations sur une pile de taille n sont : empiler **1**, dépiler **1**, vider **n**. Mais on peut compter tout les dépilements au moment d'empiler ce qui donne les coût : empiler **2**, dépiler **0**, vider **0**.

Remarque 21: On voit donc que le coût de ces opérations ne dépend que du nombre d'empilement

3) Méthode du potentiel

Ici, on considère Φ une fonction qui à chaque entrée associe une valeur dite potentielle. Le coût amortie d'une opération est donc son coût de base plus la différence de potentiel

Exemple 22: Dans le cas de la pile, Φ représente la taille de la pile. Les coûts amortis sont donc :

empiler : $1 + \Phi(n+1) - \Phi(n) = 2$ dépiler : $1 + \Phi(n-1) - \Phi(n) = 0$ vider : $n + \Phi(0) - \Phi(n) = 0$

IV Amélioration de la complexité

1) Calibrage temps/espace

Selon l'utilisation de l'algorithme, il peut être intéressant d'augmenter la complexité spatiale pour améliorer la complexité temporelle et vice versa.

Exemple 23 : Le calcul du nème terme de Fibonacci par un algorithme récursif est meilleur en espace qu'un algorithme par programmation dynamique, mais moins bon en temps :

	temps	espace
récursif	$O(\varphi^n)$	$O(1)$
prog. dynamique	$O(n)$	$O(n)$

Remarque 24: En pratique, il faut aussi faire attention car une trop grosse complexité en espace peut avoir des répercution temporelle si les cases mémoires sont trop éloignées les unes des autres.

2) Choix des structures de donnée

Selon la structure de donné choisie, les opérations fondamentales n'ont pas le même coût.

Exemple 25: L'algorithme de Dijkstra, qui permet de calculer le plus court chemin entre deux sommet d'un graphe a une complexité $O(n(n+\delta))$ lorsqu'on utilise des tableaux et une complexité $O(n\delta \log n)$ si on utilise des tas binaires. Où δ est l'arité maximale du graphe et n sa taille [DEV]

- Algo de Dijkstra :

Hypothèses de l'algo ?

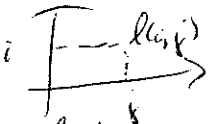
-> graphe connexe.

-> pas de poids négatifs.

sinon, Bellman-Ford ;
pas de cycle de poids négatif.

Différentes représentations de graphe ?

$G = (S, A)$; $|S| = n$ => matrice $n \times n$.



-> tableau de listes d'adjacence.

$[0, \dots, n-1]$

↓

contient les listes des voisins de 0.

Autres manières d'utiliser Dijkstra ?

Ex tas de Fibonacci : on améliore la complexité amortie.

Clou : Qu'est-ce qui peut modifier une complexité moyenne ?

↳ un cas très coûteux a un poids plus fort.

→ Des hypothèses sur l'entrée.

Des hypo sur l'entrée ont-elles un effet sur la complexité amortie ?

Non.

Dijkstra

Soit $G = (V, E)$ un graphe orienté, $l : E \to \mathbb{N}$ la longueur des arcs de G et $dist : V \times V \to \mathbb{N} \cup \{\infty\}$ la longueur des plus courts chemins entre deux nœuds.

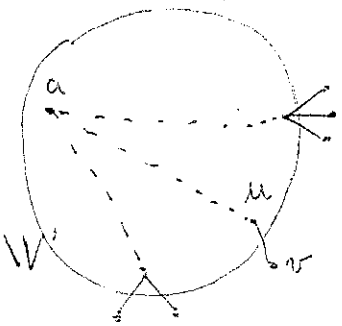
Pour $a \in V$, on veut calculer $d : V \to \mathbb{N} \cup \{\infty\}$ tq $\forall u \in V$, $d(u) = dist(a, u)$

Dijkstra (G, l, a)

- **Init :** $d(a) = 0$
  - $\forall u \in V \setminus \{a\}$, $d(u) = \infty$
  - $V' \leftarrow V \setminus \{a\}$
- **Tantque** $V' \neq \emptyset$:
  - Trouver $v \in V'$ tq $d(v)$ soit minimal — Extraire_min
  - $V' \leftarrow V' \setminus \{v\}$
  - **Pour tous** $(v, z) \in E$: — Trouver_successeurs
    - **si** $d(z) > d(v) + l(v, z)$:
      - $d(z) \leftarrow d(v) + l(v, z)$ — Diminuer_clé
- **Retourner** d

Grâce à tas

$R = V \setminus V'$



Preuve par induction sur $R = V \setminus V'$

(H_i) :
- (1) $\exists k$ tq $\forall u \in R$, $dist(a, u) \leq k$ et $\forall u \in V'$, $dist(a, u) > k$.
- (2) $\forall u \in V$, $d(u)$ est la longueur du plus court chemin entre a et u dont tous les sommets, sauf éventuellement u, sont dans R.

Cas de base immédiat

Cas inductif : $R' = R \cup \{v\}$ tq $d(v) = \min_{u \in V'} d(u) \to (H_1)$ ok

Soit $u \in R$, $d'(u) = u$
Soit $u \in V \setminus R$ tq $\exists u \in R$ et $(u, u) \in E$: $d'(u) = \min_{\substack{u \in R \\ (u,u) \in E}} d(u) + l(u, u)$ $\Big\}$ (H_2) ok

Soit, $d'(u) = \ldots$

Analyse de complexité :

Soient $n = |V|$, δ l'arité maximale de G.

On suppose qu'on calcule l en temps constant

Alors $C($ Dijkstra$(G, l, a)) \leq C($ créer_tas$(n))$

$+ \quad n \times \left[C(\text{Extraire\_min}(n)) + C(\text{Successeurs}) + \delta\, C(\text{Diminuer\_clé}(n)) \right]$

Implémentation :

Pour G . Matrice $n \times n$ $\quad C(\text{Succ}) = O(n)$

. Liste d'adjacence . $C(\text{Succ}) = O(1)$

Pour le Tas :	$C($créer_tas$)$	$C($Extraire_min$)$	$C($Diminuer_clé$)$
Tableau (non-trié)	$O(n)$	$O(n)$	$O(1)$
Tas binaire	$O(n \log n)$	$O(\log n)$	$O(\log n)$

Coût Total

Tableau : $O(1) + n(O(n) + \delta \times O(1)) = O(n(n+\delta))$

Tas binaire : $O(n \log n) + n(O(\log n) + \delta\, O(\log n)) = O(n \delta \log n)$

$\longrightarrow$ Si δ est proche de n, il vaut mieux prendre un tableau !

Tri polyphasé

Mathias Millet

2014

Introduction

L'algorithme du tri polyphasé est un algorithme de tri adapté de l'algorithme de tri fusion. Il permet d'obtenir de bonnes performances dans le cas d'utilisation de bandes magnétiques en tant que mémoires externes.

Nous utiliserons ici la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$, définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$.

Modèle de calcul

On se place dans le modèle suivant :

- Un processeur avec une mémoire centrale de taille M
- Trois bandes magnétiques pour stoquer les données, chacune équipée d'une tête de lecture qui peut lire ou écrire une case mémoire à la fois. On supposera les bandes de longueur non bornée.

On dispose donc des opérations élémentaires supplémentaires suivantes :

- `var` $\leftarrow$ `lire`$_i$: donne à la variable `var` la valeur lue sur la bande i ;
- `écrire`$_i$(`var`) : écris la valeur de `var` sur la bande i ;
- `avancer`$_i$: avance d'une case mémoire sur la bande i ;
- `reculer`$_i$: recule d'une case mémoire sur la bande i ;

Complexité Entrées/Sorties. Dans ce modèle de calcul, on peut définir la complexité entrées/sorties d'un algorithme comme le nombre d'opérations `avancer` et `reculer` effectuées pendant une exécution.

Tri polyphasé

Définition (Monotonie) Une monotonie est un tableau trié par ordre croissant, placé sur une bande magnétique.

Définition (Fusion) La fusion de deux monotonies M_1 et M_2, de taille m_1 et m_2 respectivement, est la création d'une nouvelle monotonie, contenant exactement la réunion des éléments de M_1 et M_2. L'algorithme de la fusion est directement adapté de celui du tri fusion. Pour être efficace en terme d'entrées/sorties, chaque monotonie doit se trouver sur une bande différente, et les têtes de lecture doivent être placées initialement au début des monotonies. La complexité de l'opération de fusion est alors en $O(m_1 + m_2)$.

On suppose en général que les données lues sont effacées au fur et à mesure.

Algorithme du tri polyphasé

On suppose que le tableau à trier se trouve sur une seule bande. La taille de ce tableau est N. On effectue les étapes suivantes :

- On trouve n tel que $\frac{N}{F_n} > M \geq \frac{N}{F_{n+1}}$.
- On découpe les données en paquets de taille $t = \frac{N}{F_{n+1}}$ (on obtient donc F_{n+1} paquets), qu'on trie en mémoire, puis qu'on replace sur les autres bandes, obtenant ainsi des monotonies. On en place F_{n-1} sur la première bande, F_n sur la deuxième.
- On fusionne alors les paquets comme expliqué ci-dessous.

Fusion des monotonies

Première étape La première étape se déroule de la manière suivante :

1. La première bande contient F_n monotonies de taille t ;
2. La deuxième bande contien F_{n-1} monotonies de taille t ;
3. La troisième bande est vide.

On fusionne alors 2 à 2 les monotonies des deux premières bandes, en plaçant les nouvelles monotonies obtenues sur la troisième bande. On effectue ainsi F_{n-1} fusions, de complexité $2t$ chacune.

On obtient à la fin des fusions :

1. La première bande contient F_{n-2} monotonies de taille $t \cdot F_{i+1}$;
2. La deuxième bande est vide.
3. La troisième bande contient F_{n-1} monotonies de taille $t \cdot F_{i+2}$.

$i^{\text{ème}}$ étape Au début de la $i^{\text{ème}}$ étape, on a la situation suivante :

1. Une bande avec F_{n-i+1} monotonies de taille $t \cdot F_i$;
2. Une bande avec F_{n-i} monotonies de taille $t \cdot F_{i+1}$;
3. Une bande vide.

On fusionne alors 2 à 2 les monotonies des deux premières bandes, en plaçant les nouvelles monotonies obtenues sur la troisième bande. On effectue ainsi F_{n-i} fusions, de complexité $t \cdot F_i + t \cdot F_{i+1}$ chacune.

On obtient à la fin des fusions :

1. Une bande avec F_{n-i-1} monotonies de taille $t \cdot F_{i+1}$;
2. Une bande vide ;
3. Une bande avec F_{n-i} monotonies de taille $t \cdot F_{i+2}$.

Arrêt de l'algorithme On obtient une unique monotonie à la fin de la $(n-1)^{\text{ème}}$ étape. Les données sont alors triées !

Calcul de complexité

Théorème : La complexité E/S de l'algorithme du tri polyphasé est en $O\left(N \log\left(\frac{N}{M}\right)\right)$.

Preuve : Le tri et la formation des monotonies initiales sont faits en complexité linéaire. Calculons maintenant la complexité des fusions de monotonies. La fusion s'exécute en $n-1$ étapes, la $i^{\text{ème}}$ étape impliquant F_{n-i} fusions de complexité $O(t \cdot F_i + t \cdot F_{i+1})$ chacune, d'où une complexité totale en

$$O\left(t \cdot \sum_{i=1}^{n-1} F_{n-i} F_{i+2}\right)$$

. On dispose aussi des deux lemmes suivants :

Lemme 1 : $\displaystyle\sum_{k=0}^{n} F_{n-k} F_k = \frac{n-1}{5} F_n + \frac{2n}{5} F_{n-1}.$

Lemme 2 : $F_n = \frac{1}{\sqrt{5}}(\phi^n + \bar{\phi}^n)$ avec $|\phi| > 1$, $|\bar{\phi}| < 1$.

Mettons notre expression sous une forme compatible avec celle du lemme 1 :

$$C = t \cdot \sum_{i=1}^{n-1} F_{n-i} F_{i+2} = t \cdot \sum_{i=1}^{n-1} F_{n+2-(i+2)} F_{i+2} = t \cdot \sum_{k=3}^{n+1} F_{n+2-k} F_k$$

$$= t \cdot \left(\sum_{k=0}^{n+2} F_{n+2-k} F_k \quad - \quad (2 \cdot F_0 F_{n+2} + F_1 F_{n+1} + F_2 F_n) \right)$$

On peut maintenant réécrire

$$C = t \cdot \left(\frac{n+1}{5} F_{n+2} + \frac{2(n+2)}{5} F_{n+1} - F_{n+2} \right) = t \cdot \left(\frac{n-4}{5} F_{n+2} + \frac{2(n+2)}{5} F_{n+1} \right)$$

Or on sait que $t = \dfrac{N}{F_{n+1}}$, on a donc $C = N \cdot \left(\dfrac{n-4}{5} \dfrac{F_{n+2}}{F_{n+1}} + \dfrac{2(n+2)}{5} \right)$

Par définition de $(F_n)_n$, on a $\dfrac{F_{n+2}}{F_{n+1}} = 1 + \dfrac{F_n}{F_{n+1}} = O(1)$, et

$$\log(F_n) = -\frac{1}{2}\log(5) + n\log(\phi) + \log(1 + o(1)) \Longrightarrow n = O(\log(F_n)).$$

On obtient donc que $C = O(N \log(F_n))$.

Finalement, comme $\dfrac{N}{F_n} > M$, $F_n = O(\frac{N}{M})$, et la complexité de l'algorithme est

$$\boxed{O\left(N \log(\frac{N}{M})\right)}$$

□

Preuve du lemme 1 : Montrons par récurrence, pour $n \geq 1$, la propriété

$$\mathcal{P}(n) : \quad \sum_{k=0}^{n} F_{n-k} F_k = \frac{n-1}{5} F_n + \frac{2n}{5} F_{n-1}$$

- $\mathcal{P}(1) : 2F_1F_0 = 0F_1 + \frac{2}{5}F_0 = 0$ est vraie.
- $\mathcal{P}(2) : 2F_2F_0 + F_1^2 = \frac{1}{5}F_2 + \frac{4}{5}F_1 = 1$ est vraie.

- Soit $n \geq 2$, supposons $\mathcal{P}(n)$ et $\mathcal{P}(n-1)$ vraies. On a

$$\begin{aligned}
\sum_{k=0}^{n+1} F_{n+1-k}F_k &= \sum_{k=0}^{n-1} F_{n+1-k}F_k \quad + \quad F_nF_1 + F_{n+1}F_0 = \sum_{k=0}^{n-1}(F_{n-k}+F_{n-1-k})F_k \quad + \quad F_n \\
&= \sum_{k=0}^{n-1} F_{n-k}F_k + \sum_{k=0}^{n-1} F_{n-1-k}F_k \quad + \quad F_n = \sum_{k=0}^{n} F_{n-k}F_k + \sum_{k=0}^{n-1} F_{n-1-k}F_k \quad + \quad F_n \\
&= \frac{n-1}{5}F_n + \frac{2n}{5}F_{n-1} \quad + \quad \frac{n-2}{5}F_{n-1} + \frac{2(n-1)}{5}F_{n-2} \quad + \quad F_n \\
&= \frac{4}{5}F_n + \frac{n}{5}(F_n + F_{n-1}) + \frac{2(n-1)}{5}(F_{n-1}+F_{n-2}) \\
&= \frac{n}{5}F_{n+1} + \frac{2(n-1)}{5}F_n = \frac{4}{5}F_n + \frac{n}{5}F_{n+1} + \frac{2(n+1)}{5}F_n
\end{aligned}$$

  Donc $\mathcal{P}(n+1)$ est vraie.

On a ainsi prouvé par récurrence que, pour tout $n \geq 1$, et la propriété $\mathcal{P}(n)$ est vraie. □

Preuve du lemme 2 : D'après le théorème sur les suites récurrentes définies par une équation linéaires, les suites $(u_n)_n$ qui vérifient $u_{n+2} = u_{n+1} + u_n$ pour $n \geq 0$ sont de la forme $u_n = \lambda\phi^n + \mu\bar{\phi}^n$, où $\phi = \frac{1+\sqrt{5}}{2}$ et $\bar{\phi} = \frac{1-\sqrt{5}}{2}$ sont les deux solutions de l'équation $X^2 - X - 1 = 0$.

Les conditions initiales nous permettent de poser : $F_0 = \lambda + \mu = 0$ et $F_1 = \lambda\phi + \mu\bar{\phi} = \frac{1}{2}((\lambda+\mu) + \sqrt{5}(\lambda-\mu) = 1$, d'où $\lambda = -\mu = \frac{1}{\sqrt{5}}$, et $F_n = \frac{1}{\sqrt{5}}(\phi^n + \bar{\phi}^n)$. □

Références

- Christine Froidevaux, Marie-Claude Gaudel, Michèle Soria, *Types de données et algorithmes.*
  Directement tiré de ce livre
- Donald Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*
  Traite le tri polyphasé, je n'ai pas regardé en détails.