Objectifs: Il existe souvent plusieurs méthodes, ou algorithmes pour résoudre un même problème. On veut pouvoir les comparer : indépendamment de l'implémentation (langage, machine...)
→ Quelle contraintes considérer ?
Hypothèse : on ne considérera que des algorithmes séquentiels.

I] Introduction

Déf 1: Un algorithme décrit un traitement sur un certain nombre, fini, de données (éventuellement aucune). C'est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :
- définie de façon rigoureuse et non ambiguë
- effective c'est-à-dire pouvant être effectivement réalisée par une machine ; cela correspond à une action qui peut être réalisée avec un papier et un crayon en un temps fini.

Rq: On suppose que notre algorithme termine et fait bien ce qu'on pense qu'il fait. De plus, on étudie un algo donné avec son implémentation et ses structures de données, on ne cherche pas à l'optimiser.

Déf 2: On calcul le coût d'un algorithme en prenant en compte uniquement un ou plusieurs types d'opérations : les opérations fondamentales.

exemple 1: * algorithme de tri de tableau
↳ opérations fondamentales : comparaisons entre 2 éléments.
* multiplications de matrices
↳ opérations fondamentales : nombre d'additions et de multiplications

Rq: On ne considère que les opérations élémentaires afin de faciliter notre analyse.

Déf 3: Etant donné un algorithme A et une entrée x, on note C(x) le nombre d'opérations élémentaires effectuées lors de l'exécution de A sur l'entrée x.

Déf 4: La taille n d'une entrée est un nombre permettant de caractériser l'entrée pour un algorithme.

exemple 2: * le nombre d'éléments du tableau (algo de tri)
* le nombre de bits (eg : si l'entrée est un entier)
* la hauteur (si l'entrée est un arbre)

Déf 5: Complexité au pire cas

$$C_{max}(n) = \max_{x \text{ de taille } n} C(x)$$

Déf 6: Complexité en moyenne
On se donne la loi de probabilité selon laquelle les données sont distribuées

$$C_{moy}(n) = \sum_{x \text{ de taille } n} p(x)\, C(x)$$

où $p(x)$ est la probabilité d'avoir la donnée x parmi toutes les données de taille n.

exemple 3: * On dispose d'un tableau de taille n dont les éléments $\in [\![1,k]\!]$. On cherche s'il existe $i \in [\![1,n]\!]$ tel que $T[i] == v$, pour $v \in [\![1,k]\!]$.

```
trouve(T, v) :
    pour i=0 à T.taille
        si T[i] == v
            retourne vrai
    retourne faux.
```

$C_{max}(n) = \Theta(n)$

$C_{moy} = k\left[1-\left(1-\frac{1}{k}\right)^n\right]$ (si $v \notin T$ $C = D(n)$ sinon v apparaît pour la 1ère fois à la place i avec proba $\frac{(k-1)^{i-1}}{k^i}$)

Pb: Dans l'exemple précédent, on a supposé que tous les tableaux à n éléments dans $\{1,..k\}$ étaient équiprobables. Mais ce n'est pas toujours le cas.
contre-exemple : Recherche de motif dans un texte.
En langue française, "est" est plus fréquent que "ria" dans un texte généraliste.

II - Exemples de méthodes de calcul

1) Notations de Landau

En pratique, on n'a pas besoin d'un calcul exact mais seulement d'une bonne estimation, d'une borne supérieure ou d'une estimation asymptotique suivant les cas. Parfois, une borne inférieur est intéressante (ex: tri $O(n \log n)$)

Def 7: Soient f et g deux fonctions $\mathbb{N} \to \mathbb{R}^+$

On note:
* $f(n) = O(g(n))$ ssi $\exists c > 0$ $f(n) \leq c g(n)$ pour n assez grand.
* $f(n) = \Theta(g(n))$ ssi $f = O(g)$ et $g = O(f)$
* $f = \Omega(g)$ ssi $g = O(f)$

exemple 4:
* $f(n) = 3n^2 + 2n$ $\quad f = O(n^2)$
* $f(n) = n \log n + O(n)$ $\quad f = O(n \log n)$
* $f(n) = 4$ $\quad f = O(1)$

2) Calcul de récurrence

a) cas général

Pour analyser des algorithmes récursifs, on doit généralement résoudre des relations de récurrence sur les complexités.

exemple 5: (Diviser pour régner) Algorithme de Strassen pour la multiplication de 2 matrices carrés de taille n.

$C(n) = 7C(\lfloor \frac{n}{2} \rfloor) + \Theta(n^2)$

Théorème 8: 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 un certain $\varepsilon > 0$, 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 un certain $\varepsilon > 0$ et si $af(\lfloor \frac{n}{b} \rfloor) \leq cf(n)$ pour un certain $c < 1$ et n suffisamment grand, alors $T(n) = \Theta(f(n))$.

exemple 5': Algorithme de Strassen

$C(n) = 7C(\lfloor \frac{n}{2} \rfloor) + \Theta(n^2) \longrightarrow C(n) = \Theta(n^{\log_2 7})$

$(f(n) = \Theta(n^2)) \not\Rightarrow (f = O(n^{\log_2 7 - \varepsilon}))$ avec $\varepsilon = \log_2 7 - 2$

b) Séries génératrices solutions de récurrences

Lorsque l'on se retrouve à devoir résoudre une relation de récurrence du type: $C(n) = C(n-a) + C(n-b)$ ou $NC(n) = N(N+1) + \sum_{i \leq j \leq N} (C_{j-1} + C_{N-j})$ pour $N \geq 1$

↪ tri rapide sur des tableaux de taille $\leq n$

Formellement:
* Multiplier les deux membres de la récurrence par z^n et sommer sur n.
* Évaluer les sommes pour en tirer une equation vérifiée par la série génératrice
* Résoudre l'equation pour obtenir une formule explicite.
* Developper la série génératrice en série entière pour obtenir les coefficients (ie les termes de la suite d'origine)

Rq: Dans certain cas on multiplie par $\frac{z^n}{n!}$ pour faciliter les calculs.

exemple 6: comparaison tri fusion / tri rapide

$c_{moy} = c_{max} = O(n \log n)$ / $c_{moy} = O(n \log n)$, $c_{max} = O(n^2)$ DEV

DEV

III - Complexité amortie

1) Principe général

On effectue n fois une même opération de complexité $O(f(n))$, on peut majorer la complexité par $O(nf(n))$ mais cette estimation risque d'être grossière.

exemple 7: Compteur binaire. On considère un tableau A de taille k, rempli de 0 et de 1. On veut l'incrémenter n fois (le tableau correspond au compteur, bit de poids faible à gauche):

```
incrementer (A) : i ← 0;
                  tant que A[i] = 1 et i < k
                      A[i] ← 0; i ← i+1;
                  Si i < k alors A[i] ← 1;
```

analyse naïve : n incrémentation en $O(k)$ → $O(nk)$

2) Complexité amortie

Notons $c(i)$ le coût de la i-ème opération. On cherche $a(i)$ le coût amorti tel que $(n+1)c_{max} \geq \sum_{i=0}^{n} a(i) \geq \sum_{i=0}^{n} c(i)$.

a) Méthode de l'agrégat

On calcule $c_{max}(n)$ et on pose $c(i) = \frac{c_{max}}{n}$

exemple 8 : Compteur binaire :
A[i] n'est modifié qu'une fois sur 2^i
$\rightarrow T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq 2n \Rightarrow c_i = \frac{2n}{n} = 2$

Problème : Cette majoration est peu précise et nécessite le calcul de c_{max} !

b) Méthode comptable

Cette méthode repose sur le fait que certaines opérations ne peuvent être effectuées avant d'autres opérations.
On minimise alors les 1ère (celles qui sont effectuées après) pour maximiser le coût des autres.

exemple 9 : empiler(1) ; empiler(2) ; empiler(3) ; dépiler(1)
Pour dépiler 1, il faut l'avoir empilé et avoir dépilé tout ce qui a été empilé après lui.
$a(i) - c(i)$: crédit accordé à l'opération.

exemple 10 : Compteur binaire
$a(A[i] \leftarrow 1) = 2 \quad a(A[i] \leftarrow 0) = 0$

c) Méthode du potentiel

On définit ϕ une fonction potentiel qui associe à une entrée x_i un nombre $\phi(x_i)$ où x_i est l'état de la structure après i opérations. On a alors

$$a(i) = c_{(i)} + \phi(x_i) - \phi(x_{i-1}) \text{ et } \sum_{i=1}^{n} a(i) = \sum_{i=1}^{n} c(i) + \phi(x_n) - \phi(x_0)$$

exemple 11 : Compteur binaire
On pose $\phi(A_i)$ = nb de 1 dans A après la i-ème opération
La i-ème opération "incrémenter" change t_i éléments du tableau. Le coût réel est donc $\leq t_i + 1$

Si $\phi(A_i) = 0$ alors $\phi(A_{i-1}) = t_i = k$; si $\phi(A_i) > 0$ alors
$\phi(A_i) = \phi(A_{i-1}) - t_i + 1$
$\phi(A_i) - \phi(A_{i-1}) \leq 1 - t_i$
$\rightarrow a(i) = c(i) + \phi(A_i) - \phi(A_{i-1}) \leq 2$
$\leq t_i + 1 \quad \leq 1 - t_i$

3) Exemples

* Simuler une file avec 2 piles en effectuant les opérations en $O(1)$
* Tables dynamiques : insertion et suppression en temps amorti constant $O(1)$ [DEV]
* Calcul des bords d'un mot (= plus long sous mot propre à la fois préfixe et suffixe) $\rightarrow O(|mot|)$
(↳ Application : algorithme de Knuth-Morris-Pratt)

[DEV]

IV - Le bon compromis

Def 9 : La complexité en espace est la quantité de mémoire allouée lors de l'exécution de l'algorithme

• Allouer plus d'espace peut permettre de réduire le temps d'exécution.

ex 12 : Tableau de nombre $\in \{1, \dots n\}$, on cherche le plus petit $i \in \{1, \dots n\}$ tel que $i \notin T$, on suppose $|T| = n$.

Algo 1 : Pour i = 1..n, on cherche si $i \in T$.
Spatial : $O(1)$ / temps : $O(n^2)$

Algo 2 : T' tableau de booléen initialisé à faux.
Pour k = 1..n T'[T[k]] ← vrai.
On cherche le plus petit i tel que T'[i] = faux.
spatial : $O(n)$ / temps : $O(n)$

• À complexité temporelle égale, on choisira la plus faible complexité en espace.

/!\ Une trop grosse complexité en espace peut aussi ralentir un algorithme en pratique car l'accès en mémoire secondaire est beaucoup plus lent que l'accès en mémoire principale.

Références : R. SEDGEWICK & P. FLAJOLET, Introduction à l'analyse des algorithmes.

T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN, Introduction to Algorithms.

C. FROIDEVAUX, M.-C GAUDEL, M. SORIA, Types de données et algorithmes.

Variantes possibles : Faire une partie réduction, amélioration de la complexité.
→ choisir les structures de données les mieux adaptées

Revoir complexité amortie, calcul méthode comptable.