Complexité des algorithmes temps et espace

- Le plus souvent le temps de calcul est le plus important

- Dans le pire des cas; donc chercher un majorant

- Analyse souvent facile, souvent difficile

- Quelquefois impossible

Comme fonction de la taille des entrées notation $O(\ )$ etc.

- Le temps (ou espace) dépend des entrées; pas de majorant universel

- considérer le temps comme fonction de la taille n des entrées, en nombres, bits, caractères; chercher à majorer cette fonction

- souvent exprimé $t(n) = O(f(n))$ c'est-à-dire il existe N et c tels que si $n >= N$, alors $t(n) <= cf(n)$

- aussi o, ω, Ω, Θ

Complexité d'un algorithme, pas d'un problème

- Ici, nous considérons l'analyse du temps d'un algorithme donné

- La complexité d'un problème peut être défini (approximativement) comme la complexité du meilleur algorithme possible qui le résout

- l'analyse d'un algorithme fournit un majorant de la complexité du problème

- Trouver un minorant est, le plus souvent, beaucoup plus difficile ...

Analyses simples du texte du programme

- Une boucle de 1 à n prend temps $O(n)$

- Deux boucles imbriquées de 1 à n prennent temps $O(n^2)$, etc.

- Recherche dichotomique sur n possibilités, temps $O(\log n)$

- Algorithme d'Euclide pour $pgcd(m, n)$ (supposons $m >= n$); m remplacé en 2 itérations par $m \bmod n$, soit dans le pire des cas $[(m-1)/2]$;
  donc $O(\log n)$; est-ce vraiment possible ?

Analyses plus complexes du comportement du programme

Des boucles où le nombre d'itérations n'est pas évident

- chercher quelque chose: s'il y a une borne évidente sur le nombre de cas à considérer, ça donne un majorant mais peut-être très grossier

- processus itératif qui converge, peut-être, très lentement

- boucles très simples mais où le calcul du nombre d'itérations semble trèèès difficile

```
tantque (n>1)
   si (n~pair) alors n:=n/2;
               sinon n:=3n+1
   finsi
fintantque
```

Personne ne sait si ça s'arrête pour tout n positif!

Analyses par récurrence

- Exemple le problème 3SAT:
  expression booléenne qui est la conjonction de *clauses* de la forme $(u \; ou \; v \; ou \; w)$ où u etc. sont de variables ou négations de variables

- algorithme brut: considérer les 2^n affectations possibles des n variables

- meilleur: trois sous-problèmes : u vrai, u faux mais v vrai, u et v faux mais w vrai

- taille des sous-problèmes $n-1$, $n-2$, $n-3$

- majorant donné par $t(n) = t(n-1) + t(n-2) + t(n-3)$

- $t(n) = O(\alpha^n)$ où α est la (plus grande des) racine(s) de $x^n = x^{n-1} + x^{n-2} + x^{n-3}$, ou $1 = x^{-1} + x^{-2} + x^{-3}$ (< 2)

- (pas optimal)

Solution de (quelques) récurrences le théorème maître

Quand la récurrence a une forme régulière et simple

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^{\log_b(n)}T(1) + \ldots + \ldots \\ &= n^{\log_b(a)}T(1) + \ldots + \ldots \end{aligned}$$

le plus souvent on a un des 3 cas suivants

- $T(n) = \Theta(n^{\log_b(a)})$ si $f(n)$ (suffisamment) plus petit

- $T(n) = \Theta(n^{\log_b(a)} \log_b(a))$ si $f(n) = \Theta(n^{\log_b(a)})$

- $T(n) = \Theta(f(n))$ si $f(n)$ (suffisamment) plus grand que $n^{\log_b(a)}$ et $f(n)$ (suffisamment) plus grand que $af(n/b)$.

"suffisamment" défini exactement dans le livre
existence de cas qui ne sont pas couverts par ces 3 règles

Les problèmes NP-complets

Il existe pas mal de problèmes intéressants (une définition exacte suivra mais par exemple le problème 3SAT déjà énoncé) pour lesquels on ne sait pas s'il existe un algorithme efficace (polynomial) ou non.

- Souvent le meilleur algorithme connu est $\Theta(2^n)$ (SAT)
- Rarement $n!$ (isomorphisme de (sous-)graphes)
- Ou α^n avec $\alpha \neq 2$
- Ou $2^{\sqrt{n}}$

Mais, s'il existe un algorithme efficace pour l'un de ces problèmes, il en existe un aussi pour tous les autres!

Analyse pire cas, en moyenne, ou amortie

- Pour un algorithme seul, l'analyse dans le pire des cas est (probablement) la plus importante

- pour une fonction qui va être appelée plusieurs fois par un algorithme plus grand, c'est peut-être la moyenne

- Mais ceci ne peut donner qu'un majorant probabiliste pour l'algorithme

- Si on peut démontrer que le pire comportement de la fonction ne se produit que rarement dans le fonctionnement du program, cela donne un majorant déterministe *amorti*

- par exemple, une fonction de mise à jour d'une structure de données