Analyse d'algorithmes

Résoud-t-il le problème?

- Terminaison de l'algo : pour n'importe quelle entrée, se termine-t-il? (Oui / Non)
- Correction : " " sorties conformes au problème? (Oui / Non)

Problème : proposi° logique sur les E/S de l'algo

Est-il performant?

- Complexité en temps : combien d'opé élem. effectue-t-il?
- Complexité en espace : place en mémoire?

↳ réponse à ces deux questions donnée sous forme de suit numérique, + souvent : on donne la classe de domina° à laquelle appartient cette suite : $\Theta(\dots)$ ou $O(\dots)$

ex : tri de tableau

[1, 2, 3] → 7 opé
[3, 2, 1] → 18
[1, 7, 4, 28] → 108

} opé nbe varie selon entrée

↳ mesure de la taille d'entrée pour trouver la valeur max des opé utilisées

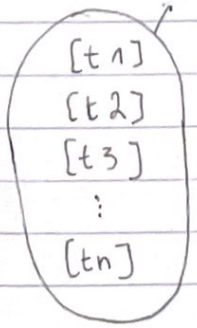
taille	nbe opé
3	18 ("pire" cas)
4	27
5	108
⋮	⋮
n	$\frac{46\, n^2}{\log n}$

} peut se caractériser en classe d'équivalence des suites

Complexité

1) Dans le pire des cas → très large & pessimiste
2) En moyenne : moy. du nbe d'opé / place occupée selon une

distribn° de proba sur les objets de taille n. souvent →
Distribn° uniforme → méthode préférée
ex: tableaux



analyse sur tous les tableaux de taille 1...n
&
on étudie le nbe opé pour chaque taille et on en fait la moyenne
(comportement quand n++ & n--)

ex: algo tri de tableaux de nbe entiers

ordre
obj formel

Entrée : un tableau d'entiers E, un ordre O sur les entiers
Sortie : un tableau d'entiers S.
Propriétée : - le tableau S soit trié selon l'ordre O
- le tableau S peut être obtenu en effectuant des échanges de valeurs entre les cases du tableau E.
Mesure : on effectue des analyses de complexités en fonction du nbe de cases du tableau E.

croissant ↗

Tri à bulles:

1er parcours

7;3;10;2;1;3;1;4 → s'il y a eu un échange, algo ♲ sinon arrêt
échange

3;7;10;2;1;3;1;4

OK

3;7;10;2;1;3;1;4

échange

3;7;2;10;1;3;1;4

⋮

3;7;2;1;3;1;4;10

↓ 2nd parcours

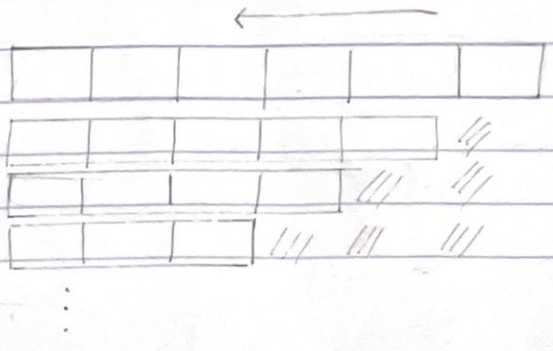
3;2;1;3;1;4;7;10

```
void triBulle(int longueur, int *T // int T[]){
    int i;
    int ech_fait = 1, temp;
    while (ech_fait) { ech_fait = 0;
        for (i=0; i<longueur; i++){
            if (T[i] > T[i+1]){
                temp = T[i];
                T[i] = T[i+1];
                T[i+1] = temp;
                ech_fait = 1;
            }
        }
    }
}
```

Bilan :

- les seules modifica° sur T sont des échanges de valeurs
  ↳ donc la propriété (2) est vérifiée
- la condi° sur "ech_fait" nous garantie que le tableau est trié lorsque l'algo se termine.
  ↳ propriété 1 est vérifiée
- à chaque tour de la boucle principale, la taille du tableau à trier diminue de 1 case au moins..
  ↳ la taille du tableau étant finie, le tableau finit par être trié.

!!! le nbe de tour boucle = longueur du tableau



Suites numériques et domination

Formellement, une suite numérique est une application de $\mathbb{N}$ vers $\mathbb{R}$.

exemple : 1, 2, 4, 8, 16, 32 ...

→ expression directe : $U_n = 2^n$ (préférée)

→ définition récursive : $U_0 = 1$, $U_{n+1} = 2 \cdot U_n$

Exemple de suites

- suite constante : $V_n = 4$ U : 4, 4, 4, 4
- suite quadratique : $H_n = n^2 = n \times n$ U : 0, 1, 4, 9, 16, 25, 36...
- suite cubique : $N_n = n^3 = n \times n \times n$ U : 0, 1, 8, 27, 64, 125...
- suite exponentielle : $U_n = 2^n = 2 \times 2 \times 2 \times 2 \dots \times 2$ U : 1 2 4 8 16 32
- suite logarithmique : $K_n = \log_2(n)$ K : Nan ($\log_2(0)$ n'existe pas, 0 1 1,58... 2 2,32... 2,58... 2,80... 3
- suite racine carrée : $L_n = \sqrt{n}$

Rappels :

- $a^b a^c = a^{b+c}$
- $(a^b)^c = a^{b \times c}$
- $\sqrt{a} = \sqrt[2]{a} = a^{1/2} \rightarrow (\sqrt{a})^2 = a = \sqrt{a^2}$

$(a^2)^{1/2} = a^{2 \times 1/2} = a^1 = a$

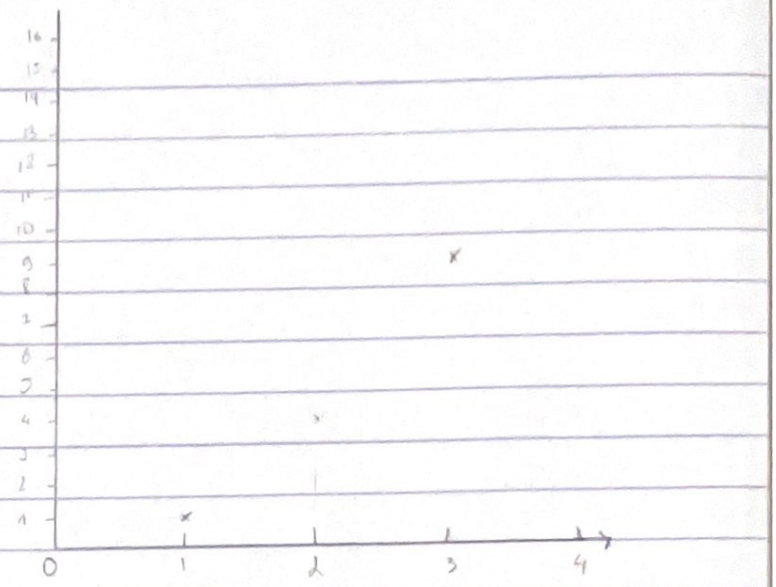
- $\sqrt[b]{a} = a^{1/b}$
- $b^{\log_b(a)} = a = \log_b(b^a)$

On peut combiner les suites de bases pour faire des suites + complexes

$V_n = n^2$ $K_n = n$ $\left(\frac{V}{K}\right)_n = \frac{V_n}{K_n} = \frac{n^2}{n} = \frac{n \times n}{n} = n$

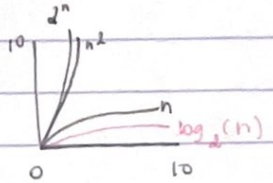
$A_n = \frac{5n^3 + 2n + 1}{n+3}$ $B = \left(\frac{n+1}{n+2}\right)^{n \times \log_{10}(n) + 5}$

n	0	1	2	3	4
$u_n = 2 \cdot n$	0	2	4	6	8
$v_n = n^2$	0	1	4	9	16

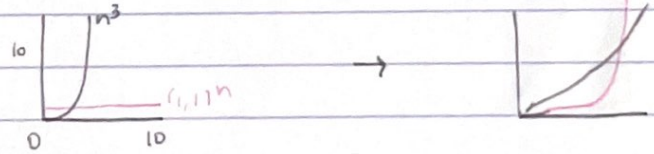


⇒ les algorithmiques les + intéressants sont ceux dont la complexité croît le + lentement

+ rapide vu que - opérations



Quelquefois les $1^{ères}$ valeurs ne suffisent pas à voir quel est la suite qui croît le + vite



Définition Domination

Soient $(V_n)_{n \in \mathbb{N}}$ & $(D_n)_{n \in \mathbb{N}}$ deux suites numériques. On dit que D domine V si

$$\exists M \in \mathbb{R}, \exists N_0 \in \mathbb{N}, \forall n > N_0, |V_n| \leq M |D_n|$$

où $|x|$ représente la valeur absolue de x.

II. valeur absolue

Remarques:

· Si la suite D ne s'annule plus au bout d'un certain terme, on a : D domine V si et ssi la suite $n \mapsto \frac{V_n}{D_n}$ est bornée

Pour deux suites quelconque, une ne domine pas forcément l'autre.

Notations

- Si D est une suite on note $O(D)$ l'ensemble des suites dominées par D.
- Si U est une suite on note $\Omega(U)$ l'ensemble des suites qui dominent U.
- $U = O(D)$ plutôt que $U \in O(D)$
- $U \ll D$ ou $U = O(D)$ ou $D = \Omega(U)$ (ou $D \gg U$).

oméga Ω

Propriétés

Soient U, V, W des suites (quelconques) et m, w des nbes réels (quelconques) alors :

- Si $U \ll V$ et $V \ll W$ alors $U \ll W$ (transitivité)
- Si $U, V \ll W$ alors $mU + V \ll W$.
- Si pour tout $n \in \mathbb{N}$, $|V_n| \leq |m| |W_n|$ alors $V \ll W$
- L'ensemble $O(1)$ est l'ensemble des suites bornées
  ↳ dominées par cette suite constante $(1, 1, 1, 1 \ldots 1)$
- L'ensemble $O(V)$ est égale à l'ensemble $|V| \cdot O(1)$
  $(= \{(V_n \cdot U_n)_{n \in \mathbb{N}} / U \text{ suite bornée}\})$

Remarques: Si U, F, W sont des suites alors

- $U \ll U$ (réflexivité)
- $O(U) O(F) = O(F.U)$
- Si $U \ll W$ alors $U.V \ll W.V$

Notation: Si U & V des suites & que $U \ll V$ & $V \ll U$ on note $U = \Theta(V)$ (ou $V = \Theta(U)$ c'est une relation d'équivalence)

thêta Θ

exemple :

$n \ll 3n$ & $3n \ll n \rightarrow 3n = \Theta(n)$

Exemples (avec n pour variable & a,b,c nbes positifs non nuls avec $c > 1$)

- $4n^5 + 3n^4 + 10n + 20 = \Theta(n^5)$
- $1 = O(n)$ mais $n \neq O(1)$
- $\log_a(n) \ll n^b \ll c^n \ll n^n$ (& pas d'égalité de classe)
- si $a < b$ alors $n^a \ll n^b$ (& pas d'égalité de classe)
  → $\frac{n^a}{n^b} = n^{(a-b)}$ → si négatif $\frac{1}{n^{|a-b|}}$ (tend vers 0) donc bornée

* pas de réciproque du genre $U \ll V$ & $V \ll U$

- si $a < b$ alors $a^n < b^n$ (& pas d'égalité de classe)
- $\log(n) = \Theta(\ln(n))$
- $n \times \ln(n) \neq O(\ln(n))$ mais $\ln(n) = O(n \ln(n))$

exemple :

$U_0 = 1$

$U_n = U_{\frac{n}{2}} + 1$ (suite réciproque)

⇒ trouver forme directe pour comparer des suites définies par récurrence

$U_{2^k} = k$

$U_{64} = U_{2^6} = U_{32} + 1$ $\qquad U_n \overset{n}{=} \log_2(n)$

$= \Theta(\log_2(n))$

$U_0 = 1$ $\qquad U_{1_6} + 1 + 1$

$U_8 + 1 + 1 + 1$

⋮

Algorithme non déterministe

- Un algo NP ne vérifie pas forcément la règle "deux exécu aux mm entrées ⇒ algo se comporte de la mm manière
- NDA (entrée 1, entrée 2...)
  ↓
  DA (entrée 1, entrée 2... suites Valeur Aléatoires) } lors du debuggage par ex
  ↳ Remarque : ne correspond pas à la complexité, infinie soit-elle

Complexité moyenne et dans le pire des cas

CM en tps (& espace) ≥ pire des cas en tps (& espace)

- $(Quicksort_n)_{n\in\mathbb{N}}$: Nbe d'opé moy du Quicksort selon n la taille du tableau
- (Q'n) : " " dans le pire des cas du Quicksort
- (F'n) : " " dans le pire des cas du Tri fusion

→ On a :

$$n\cdot\log(n) \ll Q,F' \ll n\cdot\log(n) \ll n^2 \ll Q' \ll n^2$$

Pourtant, "en pratique" QS est + efficace que le TF.
→ TF après tests est - efficace (nvelles alloca° & opéra° supplémentaires, tandis que QS ne fait que des échanges de valeurs).

Pire des cas du Qs = tableau ordonné ds un sens, on veut l'ordonner dans l'ordre inverse. n^2 complexité. (comment l'améliorer ?)
→ utiliser une valeur aléatoire comme valeur pivot. Dans la plupart des cas, ça se passe bien.

Parallélisme

Un algo est dit parallélisable si au moins deux sous-parties (sous-algo) de l'algorithme sont indépendantes en termes d'entrée-sortie.

Exemple :

```
algo para (...){
    résultat 1 : sous-algo1(...);
        ⋮
    résultat 2 : sous-algo2(...);   // entrées de cet appel ne
                                    // dépendent pas de résultat 1
}
```

Ici, les appels sous-algo 1 & sous-algo 2 peuvent être échangés "dans le temps" ou calculé/"écouté" en même tps si on implante l'algo en machine

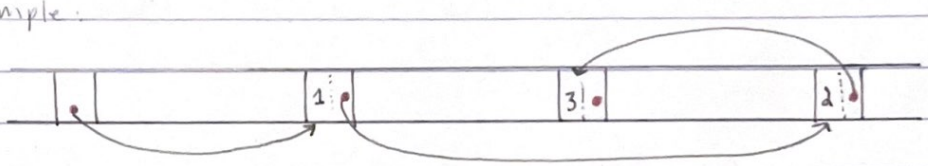
Listes chaînées

Une liste chaînée est un ensemble de zones dispersées dans la mémoire, toutes ayant la même taille et chacune contient:

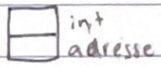
- 1 valeur
- 1 adresse vers la cellule "suivante" (logiquement)

Remarque: Dans un tableau, toutes les zones sont contiguës.

Exemple:



En C:



typedef
```
struct sCellule_t {
    int val; // valeur
    struct sCellule_t * suiv; // valeur suivante
} cellule_t;
```

Remarque: Une liste vide sera représentée par la valeur NULL.

Construction d'une liste taille 1

#include stdlib.h>
```
cellule_t * cons(int v, cellule_t * queue) {
    cellule_t * r = (cellule_t*)malloc(sizeof(*r));
    if(r == NULL) exit(1);
    (*r).val = v; // r -> val = v;
    (*r).suiv = queue; // r -> suiv = queue;
    return r;
```

(typedef val)

Appel de la fonction :

```
cellule_t* liste = NULL;
liste = cons(3, liste);
liste = cons(2, liste);
```

Fonction d'affichage :

```
void afficher (cellule_t *l)
    while (l != NULL) {
        printf("%d", (*l).val);
        l = l -> suiv;
    }
    printf("\n");
```

Fonction libération espace (pour malloc()) :

```
void libérer (cellule_t *l) {
    if (l = NULL) return;
    liberer (l -> suiv);
    free(l); l = NULL;
}
```

Exemple :