
Complexité : modèle RAM et analyse des algorithmes

M1 Informatique Luminy 2026-27

CM-TD-TP	<u>Kévin Perrot</u>	<kevin.perrot@univ-amu.fr>
TD-TP	Antonio E. Porreca	<antonio.porreca@univ-amu.fr>
TD-TP	Ravi Kunjwal	<ravi.kunjwal@univ-amu.fr>
TP	Rocco Ascone	<rocco.ascone@lis-lab.fr>

9hCM 9hTD 9hTP

$NF = \text{MAX}(ET ; 0.3 * CC + 0.7 * ET)$

Qu'est ce qu'une opération élémentaire ?

Jeu d'instructions :

https://en.wikipedia.org/wiki/X86_instruction_listings

Problèmes :

- c'est long
- valable seulement pour les processeurs x86

Machine à mémoire adressable

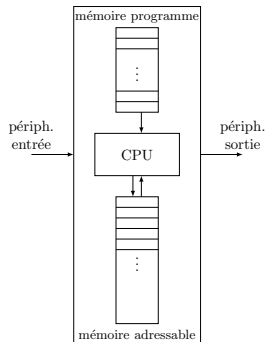
Le modèle **RAM** (Random Access Machine) est une version simplifiée d'architecture von Neumann.

[Cook Reckhow 1973]

Un programme opère avec un nombre potentiellement infini de **registres** $X_0, X_1, X_2 \dots$ pouvant chacun contenir un entier arbitraire.

↑
↓
adresses

Instructions ($i, j, k \in \mathbb{N}$)	Temps d'exécution	
	constant	logarithmique
$X_i \leftarrow C$ avec $C \in \mathbb{Z}$	1	1
$X_i \leftarrow X_j + X_k$	1	$\log X_j + \log X_k $
$X_i \leftarrow X_j - X_k$	1	$\log X_j + \log X_k $
$X_i \leftarrow X_{X_j}$	1	$\log X_j + \log X_{X_j} $
$X_{X_j} \leftarrow X_i$	1	$\log X_i + \log X_j $
TRA m si $X_j > 0$	1	$\log X_j $
READ X_i	1	$\log \text{entrée} $
PRINT X_i	1	$\log X_i $



Adressages indirects nécessaires pour qu'un programme fixé puisse accéder à un nombre non borné de registres (en fonction de l'entrée)

Universalité du modèle RAM

Théorème [Cook Reckhow 1973].

Le modèle **RAM** est **équivalent** aux **machines de Turing**.

- Une MTD qui s'exécute en temps $\mathcal{O}(T(n))$ peut être simulée par une RAM en temps $\mathcal{O}(T(n) \log T(n))$.
- Une RAM qui s'exécute en temps $\mathcal{O}(T(n))$ peut être simulée par une MTD en temps $\mathcal{O}(T(n)^2)$.

Thèse d'invariance [van Emde Boas 1990].

Tous ses modèles « raisonnables » de machines peuvent se simuler les uns les autres avec :

- un surcoût en temps polynomial, et
- un surcoût en espace qui est un facteur constant.

Analyse haut-niveau du pseudo-code

Algorithmes itératifs

Instructions élémentaires en temps $\mathcal{O}(1)$

- Déclarations de variable, affectations `int x, x=10`
- Accès à la valeur stockée dans une case de tableau `t[5]`
- Comparaison de variables de types simples `x==y, x<y`
- Opérations logiques `&&, ||, !`
- Opérations arithmétiques `+, *, /, %`



pour des entiers
pas trop grands

Branchements conditionnels : `if-then-else`

- Temps **maximum** des deux branches (pire cas)

Un algorithme sans boucle s'exécute en temps $\mathcal{O}(1)$.

Analyse haut-niveau du pseudo-code

Algorithmes itératifs (suite)

Boucles : for, while

- Avec quels outils théoriques analyse-t-on les boucles ?

Invariant de boucle (correction)

Variant de boucle (terminaison)

- **Somme** des temps de chaque itération, test inclus (+1).

↪ **Produit** si le temps de passage est le même pour chaque itération.

⇒ Il faut tâcher d'identifier le nombre d'itérations de chaque boucle.

⇒ Privilégier les boucles for plutôt que while.

Attention à la modification des itérateurs dans les boucles : c'est obscur !

Code C :

```
for(int i=0; i<100; i=i+1){  
    printf("%d\n",i);  
    i=i*2;  
}
```

Code Python :

```
for i in range(100):  
    print(i)  
    i=i*2
```

Réfléchir avant de coder :

```
int res=0; for(int i=1; i<=n; i++){ res += i; }    ≡    int res = n*(n+1)/2;  
int res=0; for(int i=0; i<n; i++){ res += 2*i+1; } ≡    int res = n*n;
```

Analyse haut-niveau du pseudo-code

Algorithmes itératifs (exemples de boucles)

pour i de 1 à n faire:

pour j de 1 à n faire:

$\mathcal{O}(1)$

↪ Complexité ? $\sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n \sum_{i=1}^n 1 = n^2$

pour i de 1 à n faire:

pour j de 1 à i faire:

$\mathcal{O}(1)$

↪ Complexité ? $\sum_{i=1}^n \sum_{j=1}^i 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$

pour i de 1 à n faire:

pour j de 1 à i*i faire:

$\mathcal{O}(1)$

↪ Complexité ? $\sum_{i=1}^n i^2 = \frac{n(2n+1)(n+1)}{6} \in \Theta(n^3)$

pour i de 1 à n faire:

pour j de 1 à i faire:

pour k de 1 à j faire:

$\mathcal{O}(1)$

↪ Complexité ? $\sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i(i+1)}{2} \in \Theta(n^3)$

pour tout nombre sur n bits faire:

$\mathcal{O}(1)$

↪ Complexité ? 2^n

pour tout $S' \subseteq S$ faire:

$\mathcal{O}(1)$

↪ Complexité ? 2^n avec $n = |S|$

pour tout $S' \subseteq S$ avec $|S'| = k$ faire:

$\mathcal{O}(1)$

↪ Complexité ? $\binom{n}{k}$ avec $n = |S|$

$(\frac{n}{k})^k \leq \binom{n}{k} \leq (\frac{ne}{k})^k \in \mathcal{O}(n^k) < 2^n$

pour toute permutation L de S faire:

$\mathcal{O}(1)$

↪ Complexité ? $n!$ avec $n = |S|$

pour i de 1 à n faire:

pour j de 1 à $\log_2(i)$ faire:

$\mathcal{O}(1)$

↪ Complexité ? $\sum_{i=1}^n \log(i) = \log(\prod_{i=1}^n i) = \log(n!)$

$\in \Theta(n \log n)$ par la formule de Stirling :

$\log_2(n!) = n \log_2 n - n \log_2 e + \mathcal{O}(\log_2 n)$

Analyse haut-niveau du pseudo-code

Algorithmes itératifs (exemple d'algorithme)

entrée : n un entier positif

$res = 0$

si $n\%2 == 0$ alors:

pour i de 1 à n faire:

$B =$ nouveau tableau de taille i

 pour j de 1 à i faire:

$B[j] = 0$

$j = 1$

$res++$

 tant que $j \leq i$ faire:

 si $B[j] == 0$ alors:

$B[j] = 1$

$j = 1$

$res++$

 sinon:

$B[j] = 0$

$j++$

sinon:

$P =$ nouveau tableau de taille n

$q = 1$

 pour i de 1 à n faire:

$P[i] = i$

$q = q*i$

 pour i de 1 à q faire:

$res++$

$j = n-1$

 tant que $P[j] > P[j+1]$ faire:

$j--$

$m = j+1$

 pour k de $j+2$ à n faire:

 si $P[m] > P[k] > P[j]$ alors:

$m = k$

 échanger $P[j]$ et $P[m]$

 pour k de 0 à $(n-j+1)/2$ faire:

 échanger $P[j+1+k]$ et $P[n-k]$

retourne res

↪ Complexité? $\mathcal{O}((n+1)!)$ et $\Omega(n!)$ car $2^n < n!$ quand $n \geq 4$.

- n pair ou impair? **maximum** dans le pire cas.

- **alors**? B compte sur i bits : $\sum_{i=1}^n \Theta(i) + \Theta(2^i) = \Theta(n^2) + \Theta(2^{n+1}) \in \Theta(2^n)$.

- **sinon**? P énumère les permutations de $\{1, \dots, n\}$: $\Theta(n) + \Theta(n!) \cdot \mathcal{O}(n) \in \mathcal{O}((n+1)!)$ et $\Omega(n!)$.

Remarque : quand vous écrivez le code, vous savez ce qu'il fait, commentez !

Analyse haut-niveau du pseudo-code

Algorithmes récurifs

Somme des temps de chaque appel récursif.

On doit résoudre une relation de récurrence.

Deux exemples :

```
let rec factorial n = match n with
```

```
| 0 -> 1
```

```
| 1 -> 1
```

```
| _ -> n * (factorial (n - 1));;
```

→ **Complexité** ? $T(n) = T(n-1) + \mathcal{O}(1)$ avec $T(1) = 1$ 1, 2, 3, 4, ... deviner $T(n) = n$
et vérifier par induction : $T(1) = 1$, $T(n) = T(n-1) + 1 = n-1 + 1 = n$ ✓ donc $\mathcal{O}(n)$.

```
let rec fibonacci n = match n with
```

```
| 0 -> 0
```

```
| 1 -> 1
```

```
| _ -> (fibonacci (n-1)) + (fibonacci (n-2));;
```

→ **Complexité** ? $T(n) = T(n-1) + T(n-2) + \mathcal{O}(1)$ avec $T(0) = T(1) = 1$ 1, 1, 3, 5, 9, 15, ...

$\leq 2 T(n-1) + \mathcal{O}(1)$ avec $T(1) = 1$ (1), 1, 3, 7, 15, ... deviner $T(n) = 2^n - 1$ ✓ donc $\mathcal{O}(2^n)$.

$\geq 2 T(n-2) + \mathcal{O}(1)$ avec $T(0) = 1$ 1, ?, 3, ?, 7, ?, 15, ...

$= 4 T(n-4) + 2 + 1 = 8 T(n-6) + 4 + 2 + 1 = 16 T(n-8) + 8 + 4 + 2 + 1$

deviner $T(n) = 2^k T(n-2k) + 2^k - 1$

vérifier $= 2^k (2 T(n-2k-2) + 1) + 2^k - 1 = 2^{k+1} T(n-2(k+1)) + 2^{k+1} - 1$ ✓

$T(0)$ atteint quand $k = \frac{n}{2}$ donc $T(n) \geq \underbrace{2^{\frac{n}{2}}}_{1} T(0) + 2^{\frac{n}{2}} - 1$ donc $\Omega(2^{\frac{n}{2}}) = \Omega(\sqrt{2}^n)$.

Analyse haut-niveau du pseudo-code

Algorithmes récurrents (suite)

Deviner et Vérifier :

<https://oeis.org/> ♥

- à partir des valeurs numériques, ou
- à partir des expressions substituées.

Récurrentes linéaires :

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k) + f(n)$$

(avec conditions initiales) si $f(n) = 0$ alors homogène sinon hétérogène

Suite arithmétique : $T(n) = T(n-1) + a$ donne $T(n) = a n + T(0) \in \Theta(n)$.

Linéaire simple : $T(n) = T(n-1) + f(n)$ donne $T(n) = \sum_{i=0}^n f(i)$.

Suite géométrique : $T(n) = a T(n-1)$ avec $a > 1$ donne $T(n) = a^n T(0) \in \Theta(a^n)$.

Linéaire homogène : $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k)$ méthode :

- deviner x^n donne $x^n = a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_k x^{n-k}$;
- diviser par x^{n-k} donne $x^k = a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_{k-1} x + a_k$;

→ équation caractéristique de la récurrence, ses solutions sont des valeurs possibles pour x ;

- une racine r de multiplicité m donne les solutions $x \in \{r^n, n r^n, n^2 r^n, \dots, n^{m-1} r^n\}$;

Théorème. Si $f(n)$ et $g(n)$ sont solutions d'une récurrence linéaire homogène, alors $h(n) = s f(n) + t g(n)$ est également solution pour tous $s, t \in \mathbb{R}$.

→ donc on doit trouver la bonne combinaison linéaire correspondant à nos conditions initiales.

On peut montrer que fibonacci est en $\Theta(\varphi^n)$ avec $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$

Analyse haut-niveau du pseudo-code

Algorithmes récurrents (suite)

OpenMIT cours "Maths for CS" chap. 10 "Recurrences"

Résoudre une relation de récurrence peut être complexe :

```
int bibi(int n, int k){  
    if(k==0 || k==n){ return 1; }  
    else{ return bibi(n-1,k-1) + bibi(n-1,k); }
```

$T(n, k) = T(n-1, k-1) + T(n-1, k) + \mathcal{O}(1)$ donne $T(n, k) \in \Theta(\binom{n}{k})$

Résoudre une relation de récurrence peut être très très complexe :

$T(n+1) = (\frac{7}{4}T(n) + \frac{1}{2}) - (\frac{5}{4}T(n) + \frac{1}{2})(-1)^{T(n)}$ itère Collatz de $T(0)$

Algorithmes basés sur le principe *diviser-pour-régner* :

$$T(n) = a T(n/b) + f(n)$$

La récursion s'arrête quand n est très petit, où l'instance est résolue en temps constant.

Deux exemples :

Recherche dichotomique : $T(n) = 1 T(n/2) + \mathcal{O}(1)$ donc $\mathcal{O}(\log n)$.

Tri fusion : $T(n) = 2 T(n/2) + \Theta(n)$ donc...

Analyse haut-niveau du pseudo-code

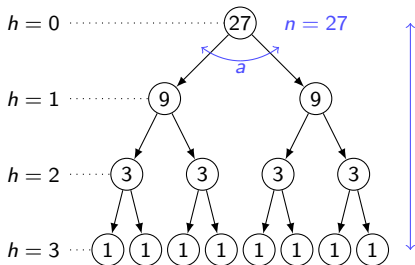
Algorithmes récursifs (suite) : diviser-pour-régner

Master théorème. Soit $a \geq 1$, $b > 1$, $c = \log_b a$, $T(n) = a T(n/b) + f(n)$.

1. Si $f(n) \in \mathcal{O}(n^{c-\epsilon})$ pour un certain $\epsilon > 0$, alors $T(n) \in \Theta(n^c)$.
2. Si $f(n) \in \Theta(n^c)$, alors $T(n) \in \Theta(n^c \log_b n)$.
3. Si $f(n) \in \Omega(n^{c+\epsilon})$ pour un certain $\epsilon > 0$, et si $a f(n/b) \leq d f(n)$ pour $d < 1$ et n suffisamment grand, alors $T(n) \in \Theta(f(n))$.

Remarque : dans certains cas le théorème ne permet pas de conclure.

Arbre des appels récursifs pour $2 T(n/3) + f(n)$:



Racine coût : $f(n) \rightsquigarrow$ cas 3

Feuilles quantité : $a^{\log_b n} = \dots = n^c$
coût : $f(1) \in \Theta(1)$
coût total : $\Theta(n^c) \rightsquigarrow$ cas 1

Noeuds de hauteur h quantité : a^h
coût : $f(\frac{n}{b^h})$

coût total de l'arbre :

$$\sum_{h=0}^{\log_b n} a^h f\left(\frac{n}{b^h}\right) = \dots \in \Theta(n^c \log_b n) \rightsquigarrow \text{cas 2}$$

$\uparrow$
 $f(n) \in \Theta(n^c)$

Analyse haut-niveau du pseudo-code

Algorithmes récurrents (exemples) : diviser-pour-régner

Master théorème. Soit $a \geq 1$, $b > 1$, $c = \log_b a$, $T(n) = a T(n/b) + f(n)$.

1. Si $f(n) \in \mathcal{O}(n^{c-\epsilon})$ pour un certain $\epsilon > 0$, alors $T(n) \in \Theta(n^c)$.
2. Si $f(n) \in \Theta(n^c)$, alors $T(n) \in \Theta(n^c \log_b n)$.
3. Si $f(n) \in \Omega(n^{c+\epsilon})$ pour un certain $\epsilon > 0$, et si $a f(n/b) \leq d f(n)$ pour $d < 1$ et n suffisamment grand, alors $T(n) \in \Theta(f(n))$.

Tri fusion : $T(n) = 2 T(n/2) + \Theta(n)$ donc...

$a = 2$, $b = 2$, $c = \log_2 2 = 1 \implies \text{cas 2} \implies T(n) \in \Theta(n \log n)$.

Recherche *diviser-pour-régner* du maximum d'un tableau T de n éléments :

maximum(T) : retourner maximum2(T,1,n)

maximum2(T,i,j) :

si $i == j$ alors retourner i

si $i+1 == j$ alors retourner $T[i] > T[j] ? i : j$

$m1 = \text{maximum2}(T, i, i+(j-i)/2)$

$m2 = \text{maximum2}(T, i+(j-i)/2+1, j)$

retourner $T[m1] > T[m2] ? m1 : m2$

$T(n) = 2 T(n/2) + \mathcal{O}(1)$ donc... $a = 2$, $b = 2$, $c = \log_2 2 = 1$, **cas 1**, $T(n) \in \Theta(n)$.

Couper en 3 la recherche du maximum ? $a = 3$, $b = 3$, $c = \log_3 3 = 1$, **cas 1**, $T(n) \in \Theta(n)$.

Théorème de Akra-Bazzi (plus général que le Master théorème).

Inexistence d'une méthode générale à suivre « bêtement »

Désolé...

Déterminer la complexité d'un algorithme est un problème **indécidable**.

Par le théorème de Rice car non-trivial (par exemple pour décider si A est $\mathcal{O}(n)$, il existe A_1, A_2 pour lesquels la réponse est « oui », « non » respectivement).

Donc il n'existe **pas de méthode automatique et toujours correcte** pour calculer la complexité d'un programme.

⇒ En général, vous pourrez analyser la complexité de vos programmes.

⇒ Pour l'analyse de complexité (et de correction) on a intérêt à garder un **code clair, organisé, commenté**.

⇒ attention aux instructions coûteuses cachées ! à éviter quand on étudie

```
if tab.contains(i) then      tab.sort()      int m = tab.max()
si i appartient à Q alors    si M ne s'arrête pas alors...
```

Compilateurs, comparatif

Les compilateurs réalisent des optimisations qui peuvent améliorer le temps de calcul, en général à un facteur multiplicatif près :

- Vectorization (plusieurs opérations en une)
- Loop unrolling (économise les manipulations d'indices)
- Loop-invariant code motion (sortir des lignes de code des boucles)
- ...

https://en.wikipedia.org/wiki/Optimizing_compiler#Common_themes

Comparatif des langages de programmation (temps, énergie, mémoire)

<https://haslab.github.io/SAFER/scp21.pdf> (page 16)

<https://doi.org/10.1016/j.scico.2021.102609>

$$a^{\log_b n} = (b^{\log_b a})^{\log_b n} = n^{\log_b a} = n^c$$

$$\sum_{h=0}^{\log_b n} a^h f\left(\frac{n}{b^h}\right) = \sum_{h=0}^{\log_b n} a^h \alpha\left(\frac{n}{b^h}\right)^c = n^c \sum_{h=0}^{\log_b n} \alpha\left(\frac{a}{b^c}\right)^h = n^c \sum_{h=0}^{\log_b n} \alpha\left(\frac{a}{a}\right)^h = n^c \alpha \log_b n$$

$$\mathbf{res} = 2^{n+1} + n!$$