Complexité des algorithmes

- Problème d'optimisation : représenté par un programme d'optimisation composé de la fonction objectif et d'un ensemble de contraintes.
- Algorithme exact : permet de traiter tous les cas possibles afin de choisir la solution la moins coûteuse.
  → consommation de ressources temporelles et matérielles. (suite à l'explosion combinatoire).
- Algorithme approché : obtenir une solution non précise dans un temps d'exécution raisonnable.

Pourquoi la complexité ?
comparer les algorithmes solutions afin de les classifier.

Comment choisir le meilleur algorithme ?
- Convergence : trouver une solution.
- Validité : solution validée.
- Optimisation des ressources :
  . spatiales : mémoire utilisée.
  . temporelle : temps d'exécution nécessaire.

au niveau de ce cours, on s'intéresse aux ressources temporelles.

Comment estimer la complexité ?
- Méthode expérimentale (approche statistique) : exécuter le programme en utilisant plusieurs valeurs de données pour observer l'évaluation du temps en fonction de la quantité de données (courbe).
- Approche formelle (approche théorique) : le coût représente le temps d'exécution théorique.
  Il ne dépend pas de l'environnement d'exécution.
  Calculé en fonction du nombre des opérations effectués par le processeur.

- Il n'est pas possible de calculer la complexité exacte : nous calculerons son ordre de grandeur grâce à des notations asymptomatiques connues sous le nom de notation de Landau.

Méthode formelle : Notation de Landau :

- $T(n)$: coût (temps d'exécution théorique).
- O : majoration du pire cas (le plus grand nombre d'opérations)

→ $f = O(g) \Leftrightarrow \exists n_0, \exists c \geq 0$ tq $\forall n \geq 0$ $f(n) \leq c \cdot g(n)$

f est dominée asymptotiquement par g (magorée par g).

- o : $f = o(g) \Leftrightarrow \forall c \geq 0$ $\exists n_0$ tq $\forall n \geq n_0$ $f(n) \leq c \cdot g(n)$

f est négligeable devant g.

- Ω : $f = \Omega(g) \Leftrightarrow \exists n_0, \exists c \geq 0$ tq $\forall n \geq n_0$ $f(n) \geq c \cdot g(n)$

$\Leftrightarrow g = O(f)$

g est majorée par f

- Θ : $f = \Theta(g) \Leftrightarrow f = O(g)$ et $g = O(f)$

$\Leftrightarrow \exists c, d \in \mathbb{R}^*, \exists n_0$ tq $\forall n > n_0$

$d \cdot g(n) \leq f(n) \leq c \cdot g(n)$

* $O(n)$ → borne supérieure → Majoration du pire cas.
* $\Omega(n)$ → borne inférieure → Minoration du meilleur des cas.
* $\Theta(n)$ → borne exacte → plus précise que les précédentes.

Quelques règles à appliquer pour déterminer la complexité O d'un algorithme à partir du nombre d'exécution des opérations $T(n)$:

- Négliger les contraintes additionnelles
  Exemple : $T(n) = c + n^2 = O(n^2)$
- Négliger les constantes multiplicatives.
  Exemple : $T(n) = C * n^2 = O(n^2)$
- Négliger les termes d'ordre inférieur.
  Exemple : $T(n) = c_1 \times n^2 + c_2 \times n^2 = O(n^2) + O(n) = O(n^2)$
- En cas de multiplication :
  Exemple : $O(n^2) \times O(n^3) = O(n^5)$

Chaque traitement élémentaire a un coût = 1 :

- La déclaration .
- L'affectation
- L'opération élémentaire
- Le test
- Le return

ont un coût = 1

Pour une boucle :

- calculer le coût total interne .
- le multiplier par le nombre d'itération .

→ Coût total = somme des coûts

* Lors du calcul du pire cas :
  s'il y a 2 blocs (ex : ~~cas~~ if et else) , on choisit le coût le plus élevé .

Les classes de complexité :

- $O(\log(n))$: Les algorithmes sub-linéaires.
- $O(n)$: Les algorithmes linéaires.
- $O(n \log(n))$: Les algorithmes quasi-linéaires.

→ rapide

- $O(n^k)$: Les algorithmes polynomiaux $(k > 1)$. → lente
- $O(a^n)$: Les algorithmes exponentiels. → impraticable.

Coût d'une opération de puissance n^k :

→ $\sum_{k=0}^{n} k$ fois càd $\frac{n(n+1)}{2}$ fois

La complexité devient $T(n) = c_1 * (n \frac{n+1}{2}) = O(n^2)$

c'est le cas d'un algorithme Trivial.

Récursivité et complexité des algorithmes récursifs

- La récursivité permet d'écrire des algorithmes plus précis et plus clairs.

⊖ Ils sont plus coûteux en terme de complexité par rapport à un algorithme itératif.

Il faut être sûr qu'on tombera toujours sur un cas d'arrêt.

⚠ On parle de récursivité terminale et non terminale que pour la récursivité simple.

- Récursivité terminale :

  L'appel récursif constitue la dernière instruction.

- Récursivité non terminale :

  Il existe d'autre(s) instruction(s) après l'appel récursif.

Exemple (Récursivité non terminale) :

```
void Algo (int n)
{ if (n>0)                    (Arrêt lorsque n=0)
  { algo (n-1);
    printf ("%d", n); } }
```

pile d'exécution

on suppose que $n = 5$:

on ne pourra pas exécuter l'instruction printf car l'exécution de l'appel récursif n'est pas encore terminée.

→ le compilateur sauvegarde cette valeur dans une pile d'exécution.

A la remontée, l'algorithme va afficher 1 2 3 4 5 .

Calcul de la complexité des algorithmes récursifs (Récursivité simple)

Exemple : Fonction puissance

```
int Puissance (x, n)
{  if (n==0) then
      return 1 ;
   else
      return (x * Puissance (x, n-1)); }
```

$\underbrace{x * Puissance(x, n-1)}_{\text{opération de base}}$

$T(n) = 2 + T(n-1)$

Calcul de la complexité des algorithmes récursifs (Récursivité multiple)

Exemple : Problème de ~~Hanoï~~ Suite de Fibonacci

```
int Fibonacci (n)
{ if (n == 0 || n == 1)        → 3
    return 1;    → 1
  else
    return (Fibonacci (n - 1) + Fibonacci (n - 2)); }
```

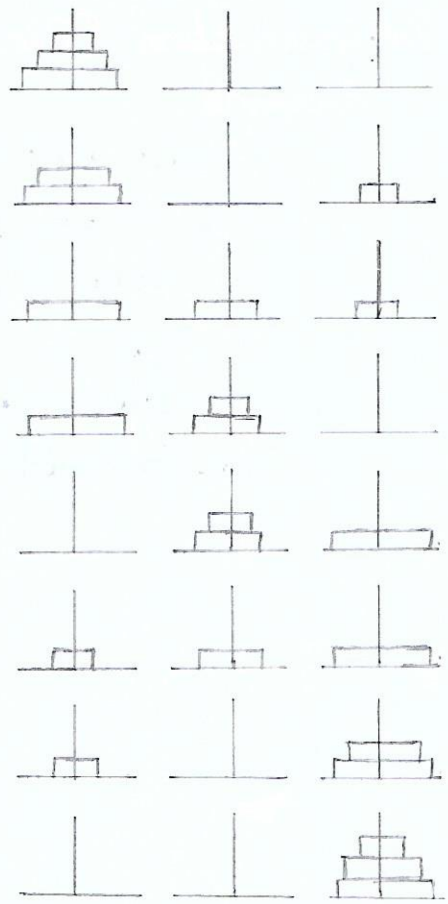
→ $T(n) = T(n-1) + T(n-2) + 4$

2 appels récursifs à 2 paramètres différents → Récursivité multiple.

Exemple : Les tours de Hanoï (Récursivité simple)

Algorithme

```
Hanoi (n, dep, int, dest)
  if (n == 1) then
     déplacer le disque de dep. vers dest.
  else
     Hanoi(n-1, dep, dest., int)
     déplacer le disque de dep. vers dest.
     Hanoi(n-1, int, dep, dest.)
  endif
EndHanoi
```



Complexité :

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n-1) + 1 + T(n-1) & \text{sinon} \end{cases}$$

Dérécursivation et complexité des algorithmes récursifs

Dérécursivation : transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

Exemple : a est un diviseur de b ? (Dérécursivation : Récursivité terminale)

```
Diviseur (int a, int b)
    if (a <= 0) then Erreur
    else
      if (a >= b) return (a == b)
      else
        return (Diviseur(a, b-a))
EndDiviseur
```

devient

```
Diviseur (int a, int b)
    if (a <= 0) then Erreur
    while (a < b) do
       b ← b - a
    end-while
    return (a == b);
End-Diviseur
```

Dérécursivation (Récursivité non terminale) :

Exemple :

```
Factoriel (int n)
    if (n == 0)
      return 1;
      else
      return n * Factoriel(n-1);
End-Factoriel
```

Dérécursiver

```
Factoriel (int n)
  Pile.init()
    while (n > 1) do
      Pile.push(n)
      n = n - 1
    end-while
    F = 1;
    while (not Pile.empty) do
      Pile.pop(n)
      F = F * n;
    End-While
    Return F;
End-Algorithme
```

Résolution de l'équation de récurrence sans second membre :

$$T(n) = \alpha_1 T(n-1) + \alpha_2 T(n-2) + \dots \alpha_k T(n-k)$$

$$\hookrightarrow \quad T(n) - \alpha_1 T(n-1) + \alpha_2 T(n-2) - \dots \alpha_k T(n-k) = 0$$

1. Associer un polynome caractéristique d'ordre k :

$$P(x) = x^k - \alpha_1 x^{k-1} - \alpha_2 x^{k-2} - \dots x_k .$$

2. Chercher les racines de $P(x)$.

3. La solution générale est sous la forme :

$$T(n) = A r_a^n + B r_b^n + \dots + \Pi r_m^n$$

(généralement on ne dépasse pas $k=2$ ou $k=3$).

Exemple : Equation de Fibonacci :

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 0 & \text{si } n = 1 \\ 1 + T(n-1) + T(n-2) & \text{si } n > 1 \end{cases}$$

on ajoute $(+1)$ pour chaque terme, on obtient :

$$S(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ S(n-1) + S(n-2) & \text{si } n > 1 \end{cases} \qquad \text{avec } S(n) = T(n)+1$$

Le polynome associé est : $P(x) = x^2 - x - 1$

dont les racines sont : $r_1 = \dfrac{1-\sqrt{5}}{2}$ et $r_2 = \dfrac{1+\sqrt{5}}{2}$

La solution générale est donc :

$$S(n) = A \left(\frac{1-\sqrt{5}}{2}\right)^n + B \left(\frac{1+\sqrt{5}}{2}\right)$$

$$\rightarrow S(0) = 1 \iff A + B = 1$$

$$\text{et } S(1) = 1 \iff A\left(\frac{1-\sqrt{5}}{2}\right) + B\left(\frac{1+\sqrt{5}}{2}\right) = 1$$

$$\rightarrow A = \frac{5+\sqrt{5}}{10} \quad \text{et} \quad B = \frac{5-\sqrt{5}}{10}$$

$$S(n) = \left(\frac{5+\sqrt{5}}{10}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n$$

$$\rightarrow T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Résolution de l'équation de ~~second~~ récurrence avec second membre

$$T(n) = \alpha_1 T(n-1) + \alpha_2 T(n-2) + \dots + \alpha_k T(n-k) + f(n)$$

Exemple éq de Fibonacci : $T(n) = T(n-1) + T(n-2) + 1$

cas particulier : $T(n) = \alpha_1 T(n-1) + f(n)$

Résolution :

$$\begin{aligned}
T(n) &= \alpha_1 T(n-1) + f(n) \\
&= \alpha_1 (\alpha_1 T(n-2)) + f(n) \\
&= \alpha_1^2 T(n-2) + \alpha_1 T(n-1) + f(n) \\
&= \alpha_1^2 (\alpha_1 T(n-3) + f(n-2)) + \alpha_1 f(n-1) + f(n) \\
&= \alpha_1^3 T(n-3) + \alpha_1^2 f(n-2) + \alpha_1 f(n-1) + f(n) \\
&\dots \\
&= \alpha_1^n T(0) + \alpha_1^{n-1} f(1) + \alpha_1^{n-2} f(2) + \dots + \alpha_1 f(n-1) + f(n)
\end{aligned}$$

$$T(n) = \alpha_1^n \left(T(0) + \sum_{i=1}^{n} \frac{f(i)}{\alpha_1^i} \right)$$

Exemple : Tour de Hanoï :

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 2T(n-1) + 1 & \text{si } n > 0 \end{cases}$$

on applique la formule :

$$T(n) = \alpha_1 T(n-1) + f(n) = \alpha_1^n \left(T(0) + \sum_{i=1}^{n} \frac{f(i)}{\alpha_1^i} \right)$$

$\rightarrow T(n) = 2^n \left(0 + \sum_{i=1}^{n} \frac{1}{2^i} \right)$

$\rightarrow T(n) = 2^n - 1$

c'est un algorithme de complexité exponentielle.

Si on suppose que le déplacement d'un disque dure une minute, pour résoudre un algorithme de 64 disques, il faut compter :

$T(n) = 2^n - 1$ min $= 35096{,}5$ milliards d'années.

Diviser pour régner

Principe :

Le paradigme " Diviser pour régner " donne lieu à 3 étapes à chaque niveau de récursivité :

- Diviser le problème en un certain nombre de sous-problèmes.
- Régner sur les sous-problèmes en les résolvant récursivement, ou si la taille est assez réduite, le résoudre directement.
- Combiner les solutions des sous-problèmes en une solution complète du problème initial.

La récurrence définissant le temps d'exécution d'un algorithme " diviser pour régner " se décompose suivant les étapes du paradigme de base :

1. Si la taille du problème est suffisamment réduite, $n \leq c$ pour une certaine constante c, la résolution est directe et consomme un temps constant $O(1)$.
2. Sinon, on divise le problème en a sous-problèmes chacun de taille $1/b$ de la taille du problème initial.
   Le temps d'exécution total se décompose en 3 parties :
   - $D(n)$: le temps nécessaire à la division du problèmes en s.p.
   - $aT(n/b)$: le temps de résolution des a s.p.
   - $C(n)$: le temps nécessaire pour construire la solution finale.

$$T(n) = \begin{cases} O(1) & \text{si } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{sinon} \end{cases}$$

Exemple : Recherche du maximum d'un tableau :

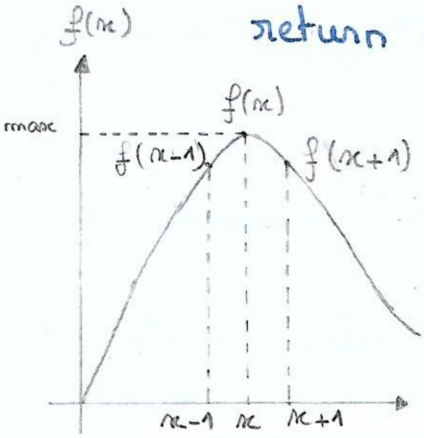
```
int Maximum (int tab[], int left, int right)
  { int k1, k2, m ;
    if (left == right)
      return tab[left] ;
    else
      m = (left + right) / 2
      k1 = maximum (x, left, m)
      k2 = maximum (x, m+1, right)
      If (k1 >= k2)
        return k1
      else
        return k2 }
```

$\Rightarrow$ $\boxed{T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2T(n/2) + 2 & \text{sinon} \end{cases}}$

Exemple : Recherche du maximum d'une fonction unimodale :
↳ un seul max

```
int maxf (int tab[], int left, int right)
  { int m = (right + left) / 2;
    if (tab[m] > tab[m+1] && tab[m] > tab[m-1])
      return tab[m] ;
    if (tab[m] > tab[m+1])
      return maxf(tab, left, m) ;
    return maxf(tab, m+1, right) ;
```

$\Rightarrow$ $\boxed{T(n) = T(n/2) + 2}$



Condition d'arrêt :

* max > f(x - 1)
  et max > f(x + 1)

Résolution des récurrences " Diviser pour régner " :

- Théorème 1 :

Soit $T(n) = a\,T(n/b) + \underbrace{f(n)}_{D(n)+C(n)}$

* Pour calculer la complexité, on doit comparer $f(n)$ à $n^{\log_b a}$ car c'est la complexité de la résolution des a sous problèmes
* $n^{(\log_b a)-\varepsilon}$ est la borne maximale de $T(n)$.

1- * si $aT(n/b) = f(n)$ alors $T(n) = \Theta(n^{\log_b a} \log n)$ (dans ce cas $f(n) = \Theta(n^{\log_b a})$)

2- * si $aT(n/b) < f(n)$:

si $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$

et $af(n/b) \leq c\,f(n)$ pour $c < 1$

alors $T(n) = \Theta(f(n))$

3- * si $aT(n/b) > f(n)$:

si $f(n) = O(n^{(\log_b a)-\varepsilon})$

alors $T(n) = \Theta(n^{\log_b a})$

- Théorème 2 : [lorsqu'on peut exprimer $f(n)$ sous forme de polynôme $(c \times n^k)$]

$$T(n) = a\,T(n/b) + \underbrace{f(n)}_{c.n^k}$$

* si $a > b^k$:

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

* si $a = b^k$:

$T(n) = \Theta(n^k \log n)$

* si $a < b^k$:

$T(n) = \Theta(n^k)$

(On compare à chaque fois $f(n)$ et $n^{\log_b a}$ et on choisit la plus grande valeur).

Exemple : Multiplication naïve ~~avec~~ de matrices :
(multiplication de matrices carrées de taille n)

algorithme naïf :

```
Multiplier (A, B, C)
  For i = 1 to n do
    For j = 1 to n do
      C(i, j) = 0
      For k = 1 to n do
        C(i, j) = C(i, j) + A(i, k) * B(k, j)
      EndFor
    EndFor
  EndFor
End_Multiplier
```

→ L'algo effectue $O(n^3)$ multiplications.

On décompose les matrices A, B et C en sous-matrices de taille $n/2 \times n/2$.
L'équation $C = AB$ peut alors se récrire :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

→ On obtient :

$$r = ae + bf \; ; \; s = ag + bh \; ; \; t = ce + df \; ; \; u = cg + dh$$

On peut donc dériver un algo diviser pour régner dont la complexité est donnée par la récurrence :

$$T(n) = 8\,T(n/2) + O(n^2)$$

4 opérations ×
$[2\,T(n/2)]$ → 2 multiplications de matrices carrées de taille $n/2$
et une addition $O(n^2)$

Puisque $a = 8$; $b = 2$ → $\log_b a = 3$; $f(n) = \Theta(n^2) = \Theta(n^{\log_b a - 1})$
car $(\log_b a) - \varepsilon = 3$ → $\varepsilon = 1$:

→ L'algorithme a donc une complexité en $O(n^3)$.

Si on applique le 2e théorème :
$f(n) = O(n^2)$ sous la forme $c \cdot n^k$ avec $c = 1$, $k = 2$
$a > b^k$ $(8 > 2^2)$ donc :
$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$