Chapitre 2 : complexité temporelle

1) Intuition

2) Taille de l'entrée, opérations élémentaires

3) Ordre de grandeur

4) Complexité en pire cas

5) Exemples

1) Intuition

- Un algorithme **ne s'exécute pas de manière instantanée.**
- Chaque instruction prend un certain temps à être exécutée, même si ce temps-là est très petit sur les ordinateurs.
- Lorsque l'on exécute notre algorithme à la main, on se bien rend compte que **certains algorithmes sont plus rapides que d'autres**, même s'ils répondent à la même question.

1) Intuition

- *Exemple:* une fonction qui prend en argument un entier n et qui renvoie la somme des entiers de 1 à *n*.

<u>Solution 1</u>

```
Fun int somme1(int n):
|    int s:=0
|    int i
|    Pour i allant de 1 à n:
|    └    s:=s+i
└    Renvoyer s
```

<u>Solution 2</u>

```
Fun int somme2(int n):
|    int s:=n*(n+1)
|    s:=s/2
└    Renvoyer s
```

- **Comptons le nombre d'affectations effectuées.**

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
    Renvoyer s
Début
    int S
    S:=somme1(4)
    print("S vaut", S)
Fin
```

Début

Variables du prog. princ.

S (int)

Affichage en console

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables du prog. princ.

S (int)

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int) 4

Nombre d'affectations jusqu'ici dans somme1: 0

Variables du prog. princ.

S (int)

Affichage en console

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	0

Nombre d'affectations jusqu'ici dans somme1: I

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	0
i (int)	

Nombre d'affectations jusqu'ici dans somme1: I

Variables du prog. princ.

S (int)	

Affichage en console

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	0
i (int)	1

Nombre d'affectations jusqu'ici dans somme1: ||

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	1
i (int)	1

Nombre d'affectations jusqu'ici dans somme1: |||

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	1
i (int)	2

Nombre d'affectations jusqu'ici dans somme1: IIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	3
i (int)	2

Nombre d'affectations jusqu'ici dans somme1: IIIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	3
i (int)	3

Nombre d'affectations jusqu'ici dans somme1: IIIIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	6
i (int)	3

Nombre d'affectations jusqu'ici dans somme1: IIIIIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	6
i (int)	4

Nombre d'affectations jusqu'ici dans somme1: IIIIIIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
            s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	10
i (int)	4

Nombre d'affectations jusqu'ici dans somme1: IIIIIIIIII

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Variables de somme1

n (int)	4
s (int)	10
i (int)	4

Valeur de retour: 10

Nombre d'affectations jusqu'ici dans somme1: ||||||||||

Variables du prog. princ.

S (int)	

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Nombre d'affectations au total dans somme1: **||||||||| =9 affectations**

Variables du prog. princ.

S (int)	10

<u>Affichage en console</u>

Exécution de la solution 1

```
Fun int somme1(int n):
        int s:=0
        int i
        Pour i allant de 1 à n:
                s:=s+i
        Renvoyer s
Début
        int S
        S:=somme1(4)
        print("S vaut", S)
Fin
```

Nombre d'affectations au total dans somme1: **||||||||| =9 affectations**

Variables du prog. princ.

S (int)	10

<u>Affichage en console</u>

S vaut 10

1) Intuition

- La solution 1 a nécessité **9 affectations**, alors que la solution 2 aurait nécessité **seulement 2 affectations.**
- Plus généralement, comptons **combien d'affectations seront faites en fonction de la valeur du paramètre *n*.**

<u>Solution 1</u>

```
Fun int somme1(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
    Renvoyer s
```

$\boxed{\text{1 aff.}}$

2 affectations à chaque fois que l'on passe dans la boucle: une pour i et une pour s

On passe n fois dans la boucle

Total: $2\mathrm{n}+1$ affectations

1) Intuition

- La solution 1 a nécessité **9 affectations**, alors que la solution 2 aurait nécessité **seulement 2 affectations**.
- Plus généralement, **comptons combien d'affectations seront faites** en fonction de la valeur du **paramètre *n***.

<u>Solution 2</u>

```
Fun int somme2(int n):
    int s:=n*(n+1)   } 1 aff.
    s:=s/2           } 1 aff.
    Renvoyer s
```

Total: 2 affectations

La solution numéro 2 a donc l'air plus rapide, surtout si l'entrée *n* devient grande!

1) Intuition

- La **fréquence d'un processeur** désigne le **nombre d'opérations qui sont effectuées en une seconde**.
- Par exemple, la fréquence du processeur de mon ordinateur est **1.8GHz**=1.8 x 10^9 Hz donc 1.8 milliards d'opérations par seconde.
- **Une opération** prend donc environ 0.5 x 10^{-9} sec, c'est-à-dire **0.5 nanoseconde**.
- Supposons que, pour résoudre un même problème (prenant en entrée un tableau de *n* entiers), on dispose :
  - d'un **Algo1 qui effectue n^2 opérations**
  - d'un **Algo2 qui effectue 2^n opérations**

1) Intuition

- Avec un processeur **1.8 GHz**: 1.8 milliards d'opérations par seconde, soit 0.5 nanoseconde par opération.

- Si n=32:

    - **Algo1** effectue n^2 = 1024 opérations, soit environ **500 nanosecondes**
    - **Algo2** effectue $2^n \approx$ 4 milliards d'opérations, soit environ **2 secondes**

- Si n=40 (seulement 8 cases de plus!):

    - **Algo1** effectue n^2 = 1600 opérations, soit environ **800 nanosecondes**
    - **Algo2** effectue $2^n \approx$ 1 000 milliards d'opérations, soit environ 500 secondes, **8 min**!!

1) Intuition

- *En résumé:* le **nombre d'opérations à effectuer** au cours de l'exécution d'un algorithme est un **critère important** pour décider si un algorithme est "bon", ou plutôt **efficace**.
- On comptera le nombre d'opérations à effectuer **en fonction d'un des éléments de l'entrée.**
- Un petite augmentation dans la taille de notre entrée peut entraîner une **grande augmentation du temps de résolution**, même avec les ordinateurs modernes rapides.

Vocabulaire

- A partir de maintenant, on utilisera souvent l'expression "**un algorithme qui prend en entrée**…"
- Cela signifie la même chose que "**une fonction qui prend en argument**…."
- *Exemple*:

Alice a écrit **un algorithme qui prend en entrée** un tableau à *n* cases et qui renvoie le maximum parmi toutes les cases du tableau.

= Alice a écrit une **fonction qui prend en argument** un tableau à *n* cases et qui renvoie le maximum parmi toutes les cases d'un tableau.

2) Que devons-nous compter, et en fonction de quoi?

Pour **compter la complexité d'un algorithme**, il faut:

- choisir en fonction de **quel paramètre de l'entrée** on va compter le nombre d'opérations
- choisir **quelles opérations** doit-on compter.

2) Que devons-nous compter, et en fonction de quoi?

Comment choisir le paramètre de l'entrée en fonction duquel on va compter le nombre d'opérations?

Ceci doit représenter **"la taille" de l'entrée**.

Pour un **tableau à *n* cases**, le paramètre choisi sera n.

Pour une **liste de longueur *n*** (voir chap. suivant), le paramètre choisi sera *n*.

En général, on le précisera dans l'énoncé.

Pour aller plus loin (facultatif)

Subtilité: si notre algorithme prend en entrée seulement **un entier *n***, alors la taille de l'entrée **n'est pas *n*.**

La taille est censée être comptée en **nombre de bits nécessaire pour écrire l'entrée.**

Or, sur *k* bits, on peut écrire les nombres de 0 à 2^k -1 (en binaire non-signé) → le nombre *n* nécessite $\log_2(n)$ bits (voire $\log_2(n)$ +1 bits) pour être écrit.

Dans ce cas, on exprimera donc le nombre d'opérations **en fonction de $k = \log_2(n)$.**

2) Que doit-on compter?

Comment choisir quelles opérations doit-on compter?

Cette question est en fait très subtile. Répondre de manière argumentée à cette question nécessite des connaissances approfondies sur les modèles de calcul, qui dépasse largement le cadre de la L1.

On précisera toujours dans l'énoncé quelles opérations on doit compter.

Pour les plus curieux: la définition exacte d'une opération élémentaire nécessite de connaître la définition d'une *Machine de Turing*, qui est un modèle de calcul abstrait, et/ou du *modèle RAM*, un autre modèle de calcul abstrait.

2) Que doit-on compter?

Exemples classiques d'opérations que l'on comptera:

- **Opération arithmétique**:
  - une addition compte pour 1 opération,
  - une multiplication compte pour 1 opération, etc....
- **comparaison d'entiers**:
  - i==j compte pour 1 opération,
  - i<j compte pour 1 opération, etc...
- **affectation**:
  - i:=0 compte pour 1 opération
- **lire une case d'un tableau, ou écrire dans une case**:
  - s:=tab[0]+5 compte pour 1 opération (lire dans tab[0])
  - tab[1]:=4 compte pour 1 opération (écrire dans tab[1])

Attention!

Une ligne de pseudo-code peut représenter plusieurs opérations!

Exemples:

- Si l'on compte les additions et les multiplications, alors la ligne **s:= n*2*m+5 compte pour 3 opérations** (deux multiplications et une addition).
- Si l'on compte les comparaisons entre entiers, la ligne de pseudo-code **Si n>3 ou n==m comporte deux opérations**.
- Si l'on compte le nombre de lectures dans un tableau, la ligne **s:=tab[0]+tab[1] compte pour 2 opérations**.

Attention!

Un **appel de fonction** peut provoquer un **grand nombre d'opérations!**

Exemple:

On reprend la fonction $\mathrm{somme1}$ du début des slides, qui provoque 2*m*+1 affectations lorsque qu'elle est appelée sur le paramètre *m*.

Supposons qu'un autre algo. contienne la ligne suivante:

$\mathrm{s:= somme1(n)+somme1(}3\mathrm{*n)}$

Cela provoque 2*n*+1 affectations pour calculer $\mathrm{somme1(n)}$

et 2*(3*n)+1 =6*n*+1 affectations pour calculer $\mathrm{somme1(}3\mathrm{*n)}$

Soit au total (2*n*+1)+(6*n*+1)+1 = **8*n*+3 affectations pour une seule ligne de pseudo-code!**

Vocabulaire: complexité temporelle

Une fois que

- l'on s'est fixé un paramètre *n* (correspondant à la taille de l'entrée)
- et que l'on a choisi les opérations que l'on doit compter,

on peut **exprimer la complexité temporelle d'un algorithme**: c'est à dire **le nombre d'opérations qui seront effectuées** au cours de l'algorithme, en fonction de *n*.

Exemple: la complexité temporelle de l'algorithme somme1 est $2n+1$.

QCM

Considérons l'algo. suivant qui prend en entrée un entier *n* et tableau à deux dimensions, supposé avoir *n* lignes et *n* colonnes.

```
Fun nombre somme2D(int n , nombre[ ][ ] tab)
|        int i,j
|        nombre s:=0
|        Pour i de 0 à n-1:
|        |        Pour j de 0 à n-1:
|        |        |        s:=s+tab[i][j]
|        |__      |__
|__      Renvoyer s
```

On choisit de compter le nombre d'**affectations** en fonction du **nombre de cases dans le tableau** (qu'on appellera *m*).

Quelle est la complexité temporelle de cet algorithme?

1) $2m + \sqrt{m} + 1$ aff., c'est-à-dire $2n^2 + n + 1$

2) 2 affectations

3) m^2 affectations

4) $3m^2$ affectations.

QCM

Considérons l'algo. suivant qui prend en entrée un entier *n* et tableau à deux dimensions, supposé avoir *n* lignes et *n* colonnes.

```
Fun nombre somme2D(int n , nombre[ ][ ] tab)
        int i,j
        nombre s:=0
        Pour i de 0 à n-1:
                Pour j de 0 à n-1:
                        s:=s+tab[i][j]
        Renvoyer s
```

1 affectation

n passages avec à chaque fois, une aff. pour j et une pour s

On choisit de compter le nombre d'**affectations** en fonction du **nombre de cases dans le tableau** (qu'on appellera *m*).

Quelle est la complexité temporelle de cet algorithme?

1) $2m + \sqrt{m} + 1$ aff., c'est-à-dire $2n^2 + n + 1$

2) 2 affectations

3) m^2 affectations

4) $3m^2$ affectations.

QCM

Considérons l'algo. suivant qui prend en entrée un entier *n* et tableau à deux dimensions, supposé avoir *n* lignes et *n* colonnes.

```
Fun nombre somme2D(int n , nombre[ ][ ] tab)
        int i,j
        nombre s:=0
        Pour i de 0 à n-1:
                Pour j de 0 à n-1:
                        s:=s+tab[i][j]
        Renvoyer s
```

1 affectation

2*n*

n passages avec à chaque fois, une aff. pour i, et 2*n* aff. pour le corps de la boucle

On choisit de compter le nombre d'**affectations** en fonction du **nombre de cases dans le tableau** (qu'on appellera *m*).
Quelle est la complexité temporelle de cet algorithme?

1) $2m + \sqrt{m} + 1$ aff., c'est-à-dire $2n^2 + n + 1$

2) 2 affectations

3) m^2 affectations

4) $3m^2$ affectations.

QCM

Considérons l'algo. suivant qui prend en entrée un entier *n* et tableau à deux dimensions, supposé avoir *n* lignes et *n* colonnes.

```
Fun nombre somme2D(int n , nombre[ ][ ] tab)
    int i,j
    nombre s:=0                      1 affectation
    Pour i de 0 à n-1:
        Pour j de 0 à n-1:
            s:=s+tab[i][j]
    Renvoyer s
```

n x $(2n+1)=2n^2+n$

On choisit de compter le nombre d'**affectations** en fonction du **nombre de cases dans le tableau** (qu'on appellera *m*).

Quelle est la complexité temporelle de cet algorithme?

1) $2m + \sqrt{m} + 1$ aff., c'est-à-dire $2n^2 + n + 1$

2) 2 affectations

3) m^2 affectations

4) $3m^2$ affectations.

QCM

Considérons l'algo. suivant qui prend en entrée un entier n et tableau à deux dimensions, supposé avoir n lignes et n colonnes.

```
Fun nombre somme2D(int n , nombre[ ][ ] tab)
        int i,j
        nombre s:=0                1 affectation
        Pour i de 0 à n-1:
                Pour j de 0 à n-1:                   2n²+n affectations
                        s:=s+tab[i][j]
        Renvoyer s
```

On choisit de compter le nombre d'**affectations** en fonction du **nombre de cases dans le tableau** (qu'on appellera m).

Quelle est la complexité temporelle de cet algorithme?

1) $2m + \sqrt{m} + 1$ **aff., c'est-à-dire** $2n^2 + n + 1$

2) 2 affectations

3) m^2 affectations

4) $3m^2$ affectations

3) Ordre de grandeur

En général, ce n'est **pas le nombre exact** d'opérations qui nous intéresse, mais **son ordre de grandeur**.

Lorsque la taille *n* de l'entrée devient très grand, il y a généralement un terme qui devient beaucoup plus important que les autres.

Exemple: Si un algorithme nécessite $\mathbf{3n^2+5n+3}$ opérations, on ne gardera que le terme de plus haut degré, c'est à dire $\mathbf{3n^2}$.

On pourra même enlever la constante multiplicative et ne garder que n^2. Dans ce cas, on écrira $\mathbf{O(n^2)}$.

3) Ordre de grandeur

Exemples:

Pour **4*n*+18** opérations, le terme le plus important (quand *n* devient très grand) est **4*n***: la **complexité temporelle** est donc **O(*n*)**.

Pour **5*n*^{3}+100*n*-45**, le terme le plus important (quand *n* devient très grand) est 5*n*^{3}: la complexité temporelle est donc **O(*n*^{3}).**

*Note: O(n) se prononce **"Grand Ô de n"***

3) Ordre de grandeur

Hiérarchie, du terme le plus important au moins important (liste non-exhaustive):

- Lorsque *n* apparaît dans l'exposant au-dessus d'une constante. *Exemples:* 2^n, $3^{n/2}$.

  On dire qu'il s'agit d'une "**exponentielle en *n***", et on le notera **$O(2^n)$**.

- Un terme **polynomial**, c'est-à-dire de la forme $c \times n^k$ où *c* et *k* ne dépendent pas de n. On le notera **$O(n^k)$**. Plus *k* est grand, plus le terme est important.

3) Ordre de grandeur

Hiérarchie, du terme le plus important au moins important (liste non-exhaustive) *(suite)*

- **Exponentiel** en *n* : **$O(2^n)$**
- **Polynomial** en *n* : **$O(n^k)$**
- Un terme **logarithmique**, c'est-à-dire de la forme **$c \times \log_2(n)$** où c ne dépend pas de *n*. On le notera **$O(\log(n))$**. ***Exemple:*** **$6 \log_2(n)$** opérations.
- Un terme **constant**, qui ne dépend pas de *n*. On le notera **$O(1)$**. **Exemple: 45** opérations.

En informatique, sauf mention explicite du contraire, le logarithme utilisé est toujours en base 2. Souvent, on le note tout simplement log(…) au lieu de $\log_2(\ldots)$.

3) Ordre de grandeur

Exemples :

- ***$2^n + 4n^3$-25***→ ordre de grandeur exponentiel en n, **$O(2^n)$**
- ***$5n^4-25n^3+7n^2$*** → ordre de grandeur "n puissance 4", **$O(n^4)$**
- **$\frac{1}{2}n^2+56n+120\log_2(n)$** → ordre de grandeur "n au carré", **$O(n^2)$**
- ***$67n+125\log_2(n) - 45$*** → ordre de grandeur "n puissance 1" ou plus simplement ordre de grandeur "n", **$O(n)$**
- ***$78\log_2(n)+210$*** → ordre de grandeur "logarithmique en n", **$O(\log(n))$.**
- ***45*** → ordre de grandeur "n puissance 0" ou plus simplement ordre de grandeur "constant", **$O(1)$.**

Ces exemples sont classés par ordre de grandeur décroissant.

3) Ordre de grandeur

Focus sur les cas les plus importants, du plus grand au plus petit:

- **$O(n^3)$**, exemple ***$26n^3+45$***. On dira que la complexité temporelle est **cubique**.
- **$O(n^2)$**, exemple ***$5n^2+72$***. On dira que la complexité temporelle est **quadratique**.
- **$O(n \log_2(n))$**, exemple ***2nlog2(n)+35n+3***. On dira que la complexité est en "n log n".
- **O(n)**, exemple ***2n+1***. On dira que la complexité est **linéaire**.
- **O(1)**, exemple ***5***. On dira que la complexité temporelle est une **constante**.

3) Ordre de grandeur

Retour sur nos exemples somme1 *et* somme2*.*

On a vu précédemment que somme1 nécessite **2n+1** affectations et somme2 seulement **2** affectations.

La complexité temporelle de somme1 est **linéaire**, notée **O(n)**.

La complexité temporelle de somme2 est **constante**, notée **O(1)**.

L'algorithme somme2 est donc **plus efficace** que l'algorithme somme1.

4) Complexité en pire cas

Il arrive souvent que le nombre d'opérations à compter ne dépende **pas seulement de la taille de l'entrée, mais aussi de ce qu'elle contient**.

Exemple: une fonction $\mathrm{cherche}$ qui prend en argument un tableau d'entier et un entier cible, et qui cherche **si cet entier apparaît dans le tableau**. Si oui, la fonction renvoie **l'indice de sa case** dans le tableau, sinon elle renvoie -1.

(voir slide suivant)

4) Complexité en pire cas

```
Fun int cherche(int tab[ ], int cible)
        int i:=0
        int aRenvoyer:=-1 /* passera à un indice valide
        du tableau lorsque l'on trouve l'entier cible */
        Tant que  i<longu(tab) et aRenvoyer==-1:
                Si tab[i]==cible:
                        aRenvoyer:=i
                i++
        Renvoyer aRenvoyer
```

4) Complexité en pire cas

Pour compter la complexité:

- On appellera ***n* le nombre de cases du tableau**, et on comptera la complexité en fonction de *n*.
- On comptera le nombre de fois où l'on **lit dans une case** du tableau.

4) Complexité en pire cas

Si le tableau contient 5, -2, 44, 10, 12, 25 et que cible vaut 10, on va lire tab[0], puis tab[1], puis tab[2], puis tab[3] → **4 lectures.**

```
Fun int cherche(int tab[ ], int cible)
    int i:=0
    int aRenvoyer:=-1 /* passera à un indice valide
    du tableau lorsque l'on trouve l'entier cible */
    Tant que  i<longu(tab) et aRenvoyer==-1:
        Si tab[i]==cible:
            aRenvoyer:=i
        i++
    Renvoyer aRenvoyer
```

4) Complexité en pire cas

Si le tableau contient 5, -2, 44, 10, 12, 25 et que cible vaut -2, on va lire tab[0], puis tab[1] → **2 lectures.**

```
Fun int cherche(int tab[ ], int cible)
        int i:=0
        int aRenvoyer:=-1 /* passera à un indice valide
        du tableau lorsque l'on trouve l'entier cible */
        Tant que  i<longu(tab) et aRenvoyer==-1:
                Si tab[i]==cible:
                        aRenvoyer:=i
                i++
        Renvoyer aRenvoyer
```

4) Complexité en pire cas

Si le tableau contient 5, -2, 44, 10, 12, 25 et que cible vaut -24, on va lire toutes les cases du tableau, sans succès → **6 lectures.**

```
Fun int cherche(int tab[ ], int cible)
    int i:=0
    int aRenvoyer:=-1 /* passera à un indice valide
    du tableau lorsque l'on trouve l'entier cible */
    Tant que  i<longu(tab) et aRenvoyer==-1:
        Si tab[i]==cible:
            aRenvoyer:=i
        i++
    Renvoyer aRenvoyer
```

4) Complexité en pire cas

Comment faire pour exprimer de façon simple la complexité temporelle de l'algorithme cherche **en fonction du nombre de cases dans le tableau?**

On comptera la complexité **"en pire cas",** c'est-à-dire qu'on dira **"au pire, on effectuera … opérations**" → on regarde le cas le moins favorable.

Sur notre exemple: **au pire, on lit toutes les cases du tableau une fois**: en effet, dans le cas le moins favorable, i va prendre toutes les valeurs entre 0 et longu(tab)-1 (c'est-à-dire entre 0 et n-1), et à chaque passage dans la boucle Tant que, on fait une lecture dans le tableau: c'est la ligne Si tab[i]==cible.

La complexité temporelle est donc n lectures en pire cas

→ **complexité linéaire,** notée **O(*n*).**

4) Plein d'exemples!

```
Fun bool contientImpair(int tab[ ])
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]%2==1:
            // si tab[i] est impair
            Renvoyer Vrai
    Renvoyer Faux
```

On comptera le **nombre de lectures dans le tableau**, en fonction de ***n* le nombre de cases dans le tableau.**

4) Plein d'exemples!

```
Fun bool contientImpair(int tab[ ])
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]%2==1:
            // si tab[i] est impair
            Renvoyer Vrai
    Renvoyer Faux
```

1 lecture

n passages dans la boucle

On comptera le **nombre de lectures dans le tableau**, en fonction de ***n* le nombre de cases dans le tableau.**

A chaque fois que l'on passe dans la boucle Pour, on lit **une fois** dans le tableau.

Il se peut que l'on quitte la fonction avec Renvoyer Vrai, mais sinon **au pire**, on passe ***n* fois dans la boucle** → **linéaire O(*n*).**

4) Plein d'exemples!

```
Fun bool contientDoublon(nombre tab[ ])
    int i,j
    Pour i de 1 à longu(tab)-1:
        Pour j de 0 à i-1:
            Si tab[i]==tab[j]:
                Renvoyer Vrai
    Renvoyer Faux
```

On comptera le **nombre de lectures dans le tableau**, en fonction de ***n*** **le nombre de cases dans le tableau**.

4) Plein d'exemples!

```
Fun bool contientDoublon(nombre tab[ ])
    int i,j
    Pour i de 1 à longu(tab)-1:
        Pour j de 0 à i-1:
            Si tab[i]==tab[j]:
                Renvoyer Vrai
    Renvoyer Faux
```

2 lectures

On comptera le **nombre de lectures dans le tableau**, en fonction de ***n*** **le nombre de cases dans le tableau**.

Peut-être que l'on quitte la fonction avec Renvoyer Vrai en plein milieu de nos deux boucles Pour, mais **au pire**, on quitte avec Renvoyer Faux.

4) Plein d'exemples!

```
Fun bool contientDoublon(nombre tab[ ])
        int i,j
        Pour i de 1 à longu(tab)-1:
                Pour j de 0 à i-1:
                        Si tab[i]==tab[j]:
                                Renvoyer Vrai
        Renvoyer Faux
```

2 lectures

Comptons le nombre de lectures dans le tableau: on en fait **2 à chaque fois** que la ligne Si tab[i]==tab[j] est exécutée.

Lorsque i vaut 1, j prend seulement la valeur 0 → 2 x 1 lecture.

Lorsque i vaut 2, j prend les valeurs 0 puis 1 → 2 x 2 lectures.

4) Plein d'exemples!

```
Fun bool contientDoublon(nombre tab[ ])
    int i,j
    Pour i de 1 à longu(tab)-1:
        Pour j de 0 à i-1:
            Si tab[i]==tab[j]:
                Renvoyer Vrai
    Renvoyer Faux
```

Au plus i passages

Au plus n passages

Lorsque *i* vaut 3, *j* prend les valeurs 0, 1 et 2 → 2 x 3 lectures.

Lorsque *i* vaut 4, *j* prend les valeurs 0, 1, 2 et 3 → 2 x 4 lectures.

Etc.. pour chaque valeur de i, ***j* prend *i* valeurs différentes** (de 0 à *i*-1), ce qui provoque **2 x *i* lectures**.

4) Plein d'exemples!

```
Fun bool contientDoublon(nombre tab[ ])
    int i,j
    Pour i de 1 à longu(tab)-1:
        Pour j de 0 à i-1:
            Si tab[i]==tab[j]:
                Renvoyer Vrai
    Renvoyer Faux
```

Au plus i passages

Au plus n passages

Donc au total, au pire, le nombre de lectures dans le tableau est:

$$2 \times 1 + 2 \times 2 + 2 \times 3 + 2 \times 4 + 2 \times 5 + \ldots + 2 \times (n\text{-}1)$$

$$= 2\,(1 + 2 + 3 + 4 + 5 + \ldots + (n\text{-}1))$$

$$= 2\,\frac{n(n-1)}{2} = n(n\text{-}1) = n^2\text{-}n = \mathbf{O(n^2)} \rightarrow \textbf{complexité quadratique}$$

4) Plein d'exemples!

```
Proc echangeAvecDernier(nombre tab[ ], int indiceAEchanger)
    nombre temp
    int n:=longu(tab)
    temp:=tab[indiceAEchanger]
    tab[indiceAEchanger]:=tab[n-1]
    tab[n-1]:=temp
```

/* Sert à échanger le contenu de la case numéro indiceAEchanger avec le contenu de la dernière case */

On comptera le **nombre de lectures et d'écritures** dans le tableau, en fonction du nombre *n* de cases dans le tableau.

4) Plein d'exemples!

```
Proc echangeAvecDernier(nombre tab[ ], int indiceAEchanger)
        nombre temp
        int n:=longu(tab)
        temp:=tab[indiceAEchanger]                  1 lecture
        tab[indiceAEchanger]:=tab[n-1]              1 lecture + 1 écriture
        tab[n-1]:=temp                              1 écriture
```

/* Sert à échanger le contenu de la case numéro indiceAEchanger avec le contenu de la dernière case */

On comptera le **nombre de lectures et d'écritures** dans le tableau, en fonction du nombre *n* de cases dans le tableau.

On fera **toujours 2 lectures et 2 écritures** dans le tableau, donc la complexité temporelle est 4: **complexité constante, notée O(1).**

4) Plein d'exemples!

```
Fun nombre rechercheMax(nombre tab[ ])
        int i
        nombre max:=tab[0]
        Pour i de 1 à longu(tab)-1:
                Si tab[i]>max:
                        max:=tab[i]
        Renvoyer max
```

On comptera le **nombre de lectures dans le tableau** en fonction de ***n*, le nombre de cases du tableau**.

4) Plein d'exemples!

```
Fun nombre rechercheMax(nombre tab[ ])
        int i
        nombre max:=tab[0]
        Pour i de 1 à longu(tab)-1:
                Si tab[i]>max:
                        max:=tab[i]
        Renvoyer max
```

1 lecture

1 ou 2 lectures

n-1 passages

On comptera le **nombre de lectures dans le tableau** en fonction de ***n*, le nombre de cases du tableau**.

On lit la **1^{e} case une fois** et **les autres cases au plus deux fois** (1 ou 2 lectures pour chaque valeur de *i* entre 1 et *n*-1) donc complexité temporelle au pire **2(*n*-1)+1=2*n*-1** → **complexité linéaire O(*n*)**

4) Plein d'exemples!

```
Fun nombre sommeTab(nombre tab[ ])
    int i
    nombre s:=0
    Pour i de 0 à longu(tab)-1:
        s:=s + tab[i]
    Renvoyer s
```

On comptera le **nombre de lectures dans le tableau** en fonction de ***n*, le nombre de cases du tableau.**

4) Plein d'exemples!

```
Fun nombre sommeTab(nombre tab[ ])
        int i
        nombre s:=0
        Pour i de 0 à longu(tab)-1:
                s:=s + tab[i]        1 lecture   } n passages
        Renvoyer s
```

On comptera le **nombre de lectures dans le tableau** en fonction de ***n*, le nombre de cases du tableau.**

On **lit chaque case exactement une fois** donc complexité temporelle égale à *n* → **complexité linéaire**, notée **O(*n*).**

Transition vers la suite

- Il existe **plusieurs manières de stocker des données**. Pour l'instant on ne connaît que les tableaux. Ils ont des avantages mais aussi des inconvénients.
- Selon les opérations que l'on devra faire souvent, il **pourra être plus "économique"** en terme de complexité temporelle de stocker nos données **différemment**.
- C'est pourquoi **il existe de nombreuses autres structures de données différentes** (listes, piles, files, arbres, tables de hachage…), à choisir au mieux en fonction de nos besoins.

Fin chapitre 2: complexité

Récapitulatif des notions qui ont été abordées:

- *Intuition sur l'efficacité/la rapidité d'un algorithme par rapport à un autre*
- *Apprendre à compter le nombre d'opérations effectuées par un algorithme.*
- *Ordres de grandeur:*
    - *Exponentiel* $O(2^n)$
    - *Cubique* $O(n^3)$ - *Quadratique* $O(n^2)$ - *Linéaire* $O(n)$
    - *Logarithmique* $O(\log(n))$
    - *Constant* $O(1)$
- *On compte la complexité du pire cas.*