Alouette, gentille alouette, alouette je te plumerai
Je te plumeras la tête, je te plumerais la tête
Et la tête, et la tête alouette, haaa

Comptine traditionnelle

9 Algorithmique

Extrait du programme

Thème : Algorithmique
Quelques algorithmes classiques sont étudiés. L'étude de leurs coûts respectifs prend tout son sens dans le cas de données nombreuses, qui peuvent être préférentiellement des données ouvertes.
Il est nécessaire de montrer l'intérêt de prouver la correction d'un algorithme pour lequel on dispose d'une spécification précise, notamment en mobilisant la notion d'invariant sur des exemples simples. La nécessité de prouver la terminaison d'un programme est mise en évidence dès qu'on utilise une boucle non bornée (ou, en terminale, des fonctions récursives) grâce à la mobilisation de la notion de variant sur des exemples simples.

Introduction

Nous avons déjà étudié un premier chapitre **Algorithmique** qui nous permis de voir les structures de base de la programmation, mais on doit aussi de poser quelques questions sur l'algorithme que nous avons conçu pour être certain qu'il fait bien ce pour quoi il a été créé.
Comme pour le premier chapitre, nous allons étudier des chansons d'un point de vue algorithmique comme l'avait fait beaucoup d'humour Donald Knuth dans l'article *Complexity of songs* disponible sur le lien suivant : `https://dl.acm.org/doi/10.1145/358027.358042`.

I. Correction d'un algorithme

Un algorithme doit donc respecter les étapes suivantes :

- On définit les **spécifications** :
  Rôle : quel est le but de l'algorithme
  Entrée(s) : quelles sont les données nécessaires pour l'exécution de l'algorithme
  Précondition(s) : quelles sont les conditions sur les données en entrée
  Sortie(s) : quelles sont les données qui sont seront retournées par l'algorithme
  Postcondition(s) : quelles sont les conditions sur les données en sortie

- On procède à la **correction partielle** :
  L'algorithme fait-il ce pour quoi il a été créé ? Autrement dit, si les entrées respectent les préconditions, est-ce que les sorties vont respecter les postconditions ?

- On étudie la **terminaison** de l'algorithme :
  autrement dit, on vérifie que l'algorithme s'arrête après un certain nombre d'étapes.

- On étudie la **complexité** de l'algorithme :
  Complexité temporelle : quel est l'efficacité (on pourrait presque dire la « rapidité ») d'exécution de l'algorithme. En première approximation, on peut l'étudier en comptant le nombre d'opérations élémentaires exécutées en fonction de la taille des données en entrée.
  Complexité spatiale : quel est la taille mémoire nécessaire à l'exécution de l'algorithme en fonction de la taille des données en entrée. Cette complexité ne sera pas étudier au lycée.
  L'énergie : pour arriver au résultat (écrire l'algorithme puis l'exécuter).
  Cette question n'est pas théorique, on pense que le numérique consomme déjà 10% de l'électricité consommée mondialement.
  `https://lejournal.cnrs.fr/articles/numerique-le-grand-gachis-energetique`.
  Cette question ne sera pas non plus étudiée au lycée.

II. Un premier exemple : *10 Green Bottles*

https://en.wikipedia.org/wiki/Ten_Green_Bottles
https://www.dailymotion.com/video/x4iikqt?syndication=273844

Voici une chanson populaire anglaise :

Ten green bottles hanging on the wall,
Ten green bottles hanging on the wall,
And if one green bottle should accidentally fall,
There'll be nine green bottles hanging on the wall.

Nine green bottles hanging on the wall,
Nine green bottles hanging on the wall,
And if one green bottle should accidentally fall,
There'll be eight green bottles hanging on the wall.

.....

One green bottle hanging on the wall,
One green bottle hanging on the wall,
And if one green bottle should accidentally fall,
There'll be no green bottles hanging on the wall.

On peut écrire un algorithme qui permet de chanter cette chanson pour n bouteilles. Par exemple, celui-ci :

Entrées : un nombre entier n
Sorties : une belle comptine

```
1  pour i allant de n à 1 faire
2  |  Chante : (i, "green bottles hanging on the wall,") ;
3  |  Chante : (i, "green bottles hanging on the wall,") ;
4  |  Chante : ("And if one green bottle should accidentally fall,") ;
5  |  si i > 1 alors
6  |  |  Chante : ("There'll be", i − 1, "green bottles hanging on the wall.");
7  |  sinon
8  |  |  Chante : ("There'll be no green bottles hanging on the wall.");
9  |  fin si
10 fin pour
```

Étudions cet algorithme, autrement dit, étudions sa spécification, sa correction partielle, sa terminaison et sa complexité (temporelle).

a) Spécifications

C'est la partie la plus simple. Elle se contente de décrire ce que doit faire l'algorithme sans se soucier du comment.

Rôle :	Créer une belle comptine.
Entrée :	Un nombre entier n.
Précondition :	$n \geq 1$.
Sortie :	Une comptine.
Postcondition :	La comptine comporte n couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à 1.

b) Correction partielle

Il faut avouer que le raisonnement ci-dessous n'est pas simple : la **correction partielle** est le point le plus délicat de ce cours. On doit pouvoir vérifier que l'algorithme effectue bien ce pour quoi il a été conçu.
Les séquences d'instructions simples (lignes 2, 3 ou 4) ne posent pas de problèmes.
Les conditionnelles (**Si ... Alors ... Sinon ...**) conduisent à un raisonnement par cas **branche alors** (ligne 6) et **branche sinon** (ligne 8).
On peut donc facilement vérifier ce que fait l'algorithme.

Le problème se situe donc dans les répétitives, autrement dit dans les boucles. Ici, on a une boucle **POUR** (lignes 1 à 10).

Pour y parvenir, on met en place un **raisonnement inductif** autour de ce qu'on appelle un **invariant**, qui :
- est un prédicat sur les variables impliquées dans la répétitive,
- est vrai juste avant la répétitive,
  (induite par la précondition et les instructions en début d'algorithme)
- est vrai après chaque itération de la répétitive
  (induite par l'invariant et condition+instructions de l'itération courante)
- sera vrai après la fin de la répétitive

La première étape consiste donc à déterminer cet invariant. Pour ce faire, on doit s'inspirer fortement de la postcondition et imaginer les différentes étapes d'exécution de l'algorithme : par rapport à la postcondition, où en serons-nous à la boucle numéro b ?
L'invariant est donc ici tout [1] trouvé :

La comptine comporte b couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+1$.

Procédons maintenant à la correction partielle. Il *suffit* de vérifier que l'**invariant** reste vrai à chaque étape de l'algorithme. Une des meilleure façon de procéder est peut-être d'utiliser un tableau.

Avant la première boucle ($b=0$)

	Valeur de i	Nombre de couplets	Invariant
Ligne 0	i n'existe pas	On a 0 couplet	La comptine comporte $0=b$ couplet **Invariant Vraie**

1. Il faut quand même penser à faire les bonnes opérations ! Le plus simple est d'imaginer que vous voulez faire $n=10$ couplets et de déterminer ce qui se passe, par exemple lorsque vous n'aurez écrit que les $b=2$ premiers. Le dernier couplet écrit contiendra le nombre $9=10-2+1$, donc la formule doit être $n-b+1$. On peut alors vérifier si cette formule semble bonne avec par exemple $b=3$.....

Première boucle ($b = 1$) : 0 couplet créé avant cette première boucle

	Valeur de i	Nombre de couplets	Invariant
Ligne 2	$i = n - b + 1 = n$	On rajoute une phrase contenant le nombre $i = n$	On a 0 couplet finalisé
Ligne 3	$i = n$	On rajoute une phrase contenant le nombre $i = n$	On a 0 couplet finalisé
Ligne 4	$i = n$	On rajoute une phrase	On a 0 couplet finalisé
Lignes 5 à 9	$i = n$	Comme $i > 1$ on rajoute une phrase contenant le nombre $i - 1 = n - 1$	La comptine comporte $1 = b$ couplet contenant le nombre entier $n-b+1 = n$ **Invariant Vraie**

Deuxième boucle ($b = 2$) : 1 couplet créé précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 2	$i = n - b + 1 = n - 1$	On rajoute une phrase contenant le nombre $i = n - 1$	On a 1 couplet finalisé contenant le nombre n
Ligne 3	$i = n - 1$	On rajoute une phrase contenant le nombre $i = n - 1$	On a 1 couplet finalisé contenant le nombre n
Ligne 4	$i = n - 1$	On rajoute une phrase	On a 1 couplet finalisé contenant le nombre n
Lignes 5 à 9	$i = n - 1$	Comme $i > 1$ on rajoute une phrase contenant le nombre $i - 1 = n - 2$	La comptine comporte $2 = b$ couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à $n - b + 1 = n - 1$ **Invariant Vraie**

...et ainsi de suite...

Oui, mais ça ne suffit pas parce que le nombre n est inconnu. Il faut faire ce type de raisonnement quelque soit la valeur de n. Pour cela il faut imaginer ce qui va se passer pour une boucle quelconque b.
Oui, mais juste avant, il devrait y avoir déjà un nombre de couplets créés. Il y en aurait ... $b-1$! Et normalement, ces $b-1$ boucles ont déjà respectées l'invariant. Ainsi on devrait avoir avant d'attaquer cette boucle b :

La comptine comporte $b - 1$ couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-(b-1)+1 = n - b + 2$

Boucle (b) : $b-1$ couplets créés précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 2	$i = n-b+1$	On rajoute une phrase contenant le nombre $i = n-b+1$	On a $b-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+2$
Ligne 3	$i = n-b+1$	On rajoute une phrase contenant le nombre $i = n-b+1$	On a $b-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+2$
Ligne 4	$i = n-b+1$	On rajoute une phrase	On a $b-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+2$
Lignes 5 à 9	$i = n-b+1$	Comme $i > 1$ on rajoute une phrase contenant le nombre $i-1 = n-b+1-1 = n-b$	La comptine comporte b couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+1$ **Invariant Vraie**

Il faut bien évidemment finir la répétitive en regardant la dernière boucle qui est la ... n-ième.

...

Dernière boucle ($b = n$) : $n-1$ couplets créés précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 2	$i = n-b+1 = 1$	On rajoute une phrase contenant le nombre $i = 1$	On a $n-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à $n-b+1 = n-n+1 = 1$
Ligne 3	$i = 1$	On rajoute une phrase contenant le nombre $i = 1$	On a $n-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à 1
Ligne 4	$i = 1$	On rajoute une phrase	On a $n-1$ couplets finalisés contenant dans chaque couplet un nombre entier strictement décroissant de n à 1
Lignes 5 à 9	$i = 1$	Comme $i \not> 1$ on rajoute la phrase de la ligne 8	La comptine comporte n couplets contenant dans chaque couplet un nombre entier strictement décroissant de n à 1 **Invariant Vraie**

Ce type de raisonnement n'est vraiment pas aisé. L'étape la plus importante est « la boucle quelconque ».
On constate que pour vérifier que l'invariant reste vrai **après** la boucle, il faut supposer qu'il est vrai **avant** la boucle.

c) Terminaison

Il faut vérifier que l'algorithme se termine bien.

- Tout algorithme sans appel de fonction ni répétitive se termine.
- Toute répétitive **POUR** se termine en un nombre fini d'étapes.

Le problème peut se poser uniquement pour les répétitives **TANT ... QUE**.
Vous vous rappelez du programme ci-contre ?

```
1 while True :
2     sing("music non strop")
```

Pour notre exemple, c'est une boucle **POUR**, il n'y a donc pas de souci, l'algorithme se termine.

d) La complexité

L'étude de la complexité d'un algorithme est une étape importante. En effet, plusieurs algorithmes différents peuvent avoir le même rôle, mais certains peuvent être plus efficaces que d'autres. Par exemple, si je veux additionner 10 fois le nombre 3, je peux soit faire :

$$3+3+3+3+3+3+3+3+3+3=30$$

soit faire

$$3\times 10=30$$

D'un côté, j'ai 10 opérations élémentaires (l'addition), de l'autre, je n'ai qu'une seule opération élémentaire (la multiplication). On pourrait aussi répondre que ce n'est plus un problème actuel car les ordinateurs modernes sont d'une très grande rapidité de calcul et faire $3+3+3+3+3+3+3+3+3+3$ ou 3×10 prend presque le même temps pour un ordinateur. Oui, mais qu'en est-il pour :

$$\boxed{\underbrace{3+3+3+3+3+3+3+...+3+3+3+3}_{1\ 000\ 000\ 000\ 000\ 000\ \text{de fois}}} \quad VS \quad \boxed{3\times 1\ 000\ 000\ 000\ 000\ 000} \quad ?$$

Il faut aussi penser que parfois, on a des contraintes physiques qui font que nous n'avons pas toujours la place ou les moyens financiers pour installer un ordinateur surpuissant (ordinateur de bord d'une voiture, sonde envoyée sur Mars, ...). Il faut donc réellement réfléchir à l'efficacité de notre algorithme.

Pour mesurer cette efficacité, on parle de **complexité**. Dans notre cas, une chose est certaine, cette complexité va dépendre du nombre n de bouteilles que l'on a au départ.
On va, en première approximation compter le nombre d'opérations élémentaires effectuées par notre algorithme.

On reprend notre algorithme :

```
1  pour i allant de n à 1 faire
2  |  Chante : (i, "green bottles hanging on the wall,") ;
3  |  Chante : (i, "green bottles hanging on the wall,") ;
4  |  Chante : ("And if one green bottle should accidentally fall,") ;
5  |  si i > 1 alors
6  |  |  Chante : ("There'll be", i − 1, "green bottles hanging on the wall.");
7  |  sinon
8  |  |  Chante : ("There'll be no green bottles hanging on the wall.");
9  |  fin si
10 fin pour
```

Lignes 1 à 10 : on répète $n-1$ fois (donc $i>1$)
  ligne 2 : 1 opération (on chante)
  ligne 3 : 1 opération (on chante)
  ligne 4 : 1 opération (on chante)
  ligne 5 : 1 opération (une comparaison $i>1$)
  ligne 6 : 2 opérations (une soustraction $i-1$, puis on chante)
Lignes 1 à 10 : on répète une dernière fois (donc $i=1$)
  ligne 2 : 1 opération (on chante)
  ligne 3 : 1 opération (on chante)
  ligne 4 : 1 opération (on chante)
  ligne 5 : 1 opération (une comparaison $i>1$)
  ligne 8 : 1 opération (on chante)

Nous obtenons donc $(n-1)\times 6+5=$ 6n -1 opérations si on souhaite avoir n couplets.

Mais ...

On pourrait objecter que certaines de ces opérations peuvent se décomposer en opérations plus élémentaires.
Par exemple, à la ligne 4, l'algorithme se contente de chanter. Ok, il y a bien une seule opération.
Mais à la ligne 2 ou 3, si on y regarde d'un peu plus près, l'algorithme regarde qu'elle est la valeur stockée en mémoire dans la variable i, rajoute cette valeur au texte puis chante la phrase obtenue. On pourrait aussi bien penser qu'il y a 3 opérations.
On pourrait aussi remarque que dans la boucle **POUR**, à la ligne 1, pour la création de la variable i on a 2 opérations : on crée un espace mémoire appelé i puis on lui affecte la valeur n.
A chaque boucle, on incrémente la variable i, ce qui compte pour 3 opérations : on regarde la valeur stockée dans i, on rajoute 1, puis on stocke le résultat dans la variable i.
Enfin, on pourrait penser que la comparaison effectuée à la ligne 5 se fait en 3 opérations : on regarde la valeur stockée en mémoire dans la variable i, on compare puis on retourne un booléen **Vrai** ou **Faux**.

Sans rentrer dans les détails du calcul, nous aurions alors $\boxed{16n}$ opérations[2].

On se demande donc si on a réellement effectuer $6n-1$ ou $16n$ opérations !

Se pose enfin une dernière question. Ces *« **opération élémentaire** »* ont-elles le même coup ? En effet, chanter un texte (ligne 4) est-il aussi simple que de faire une comparaison (ligne 5) ?
Même chose pour une soustraction et une affectation.
... etc ...
Difficile à répondre, surtout que cette réponse peut aussi dépendre du langage utilisé, de la longueur de la chaîne de caractères, ...

En réalité, nous n'avons pas besoin d'autant de détail. On peut remarquer que quelque soit la précision que nous donnerons à ce que peut être une *« **opération élémentaire** »*, la complexité s'exprimera sous la forme d'une fonction affine (nous avions $\boxed{6n-1}$ VS $\boxed{16n}$ VS ...).
Nous n'avons pas besoin d'en dire plus.

Lorsque la complexité s'exprime sous la forme d'une fonction affine, on dit que

la complexité est linéaire

et on la notera

$O(n)$

On appelle cette notation, la notation de Landau.
Si vous souhaitez en savoir plus sur l'étude de la complexité d'un algorithme et si vous aimez (énormément) les maths, vous pouvez aller visiter la page wikipédia suivante :
`https://fr.wikipedia.org/wiki/Comparaison_asymptotique`.

Cela dit, nous reviendrons très rapidement sur le sujet avec des exemples un peu plus compliqués...

2. Amusez-vous à les compter si vous voulez !

III. Un deuxième exemple : *Alouette, Gentille Alouette*

https://www.dailymotion.com/video/x1forim?syndication=273844

Voici une chanson populaire bien connue :

Alouette, gentille alouette, alouette je te plumerai
Je te plumeras la tête, je te plumerais la tête
et la tête, et la tête alouette, haaaa

Alouette, gentille alouette, alouette je te plumerai
Je te plumerai le bec, je te plumerais le bec
et le bec, et le bec, et la tête, et la tête alouette, haaaa

Alouette, gentille alouette, alouette je te plumerais
Je te plumerai le cou, je te plumerai le cou
et le cou, et le cou, et le bec, et le bec, et la tête, et la tête, alouette, haaaa

Alouette, gentille alouette, alouette je te plumerai
Je te plumerai le dos, je te plumerai le dos
et le dos, et le dos, et le cou, et le cou, et le bec, et le bec, et la tête, et la tête, alouette, haaaa

Alouette, gentille alouette, alouette je te plumerai
Je te plumerai la queue, je te plumerai la queue
at la queue, et la queue, et le dos, et le dos, et le cou, et le cou, et le bec, et le bec, et la tête et la tête, alouette, haaaa.

On peut écrire un algorithme qui permet de chanter cette chanson pour n parties du corps de l'alouette. Par exemple, celui-ci :

Entrées : un nombre entier n
Sorties : une belle comptine

```
1 T ← ['la tête', 'le bec', 'le cou', 'le dos', ...] ;
2 pour i allant de 0 à n − 1 faire
3 |   Chante : ('Alouette, gentille alouette, alouette je te plumerai') ;
4 |   Chante : ('Je te plumerai', T[i], 'je te plumerai', T[i]) ;
5 |   pour j allant de i à 0 faire
6 |   |   Chante : ('et', T[j], 'et', T[j]) ;
7 |   fin pour
8 |   Chante : ('alouette, haaaa') ;
9 fin pour
```

Étudions la spécification, la correction partielle, la terminaison et la complexité temporelle de cet algorithme.

a) Spécifications

Rôle :	Créer une belle comptine.
Entrée :	Un nombre entier n.
Précondition :	$1 \leq n \leq$ taille du tableau T.
Sortie :	Une comptine.
Postcondition :	La comptine comporte n couplets contenant dans chaque couplet de 1 à n parties distinctes du corps de l'alouette (tête, bec, cous, dos, ...).

b) Correction partielle

On regarde la postcondition et on imagine les différentes étapes d'exécution de l'algorithme : par rapport à la postcondition, où en serons-nous à la boucle numéro b ?
L'invariant est ici :

La comptine comporte b couplets contenant dans chaque couplet de 1 à b parties distinctes du corps de l'alouette.

Vérifions maintenant que l'**invariant** reste vrai à chaque étape de l'algorithme.

Avant la première boucle ($b = 0$)

	Valeur de i	Nombre de couplets	Invariant
Ligne 1	i n'existe pas	On a 0 couplet	La comptine comporte $0 = b$ couplet **Invariant Vraie**

Première boucle ($b = 1$) : 0 couplet créé avant cette première boucle

	Valeur de i	Nombre de couplets	Invariant
Ligne 3	$i = 0$	On rajoute une phrase	On a 0 couplet finalisé
Ligne 4	$i = 0$	On rajoute une phrase contenant 1 partie du corps de l'alouette	On a 0 couplet finalisé
Lignes 5 à 7	$i = 0$	On rajoute une phrase contenant 1 partie du corps de l'alouette	On a 0 couplet finalisé
Ligne 8	$i = 0$	On rajoute une phrase	La comptine comporte 1 couplet contenant dans chaque couplet 1 partie du corps de l'alouette. **Invariant Vraie**

Deuxième boucle ($b = 2$) : 1 couplet créé précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 3	$i = 1$	On rajoute une phrase	On a 1 couplet finalisé
Ligne 4	$i = 1$	On rajoute une phrase contenant 1 partie du corps de l'alouette	On a 1 couplet finalisé
Lignes 5 à 7	$i = 1$	On rajoute une phrase contenant 2 parties du corps de l'alouette	On a 1 couplet finalisé
Ligne 8	$i = 1$	On rajoute une phrase	La comptine comporte 2 couplets contenant dans chaque couplet de 1 à 2 parties du corps de l'alouette. **Invariant Vraie**

... et ainsi de suite ... que se passe-t-il pour une boucle quelconque b ?

Boucle (b) : $b - 1$ couplet créé précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 3	$i = b$	On rajoute une phrase	On a $b - 1$ couplet finalisé
Ligne 4	$i = b$	On rajoute une phrase contenant 1 partie du corps de l'alouette	On a $b - 1$ couplet finalisé
Lignes 5 à 7	$i = b$	On rajoute une phrase contenant b parties du corps de l'alouette	On a $b - 1$ couplet finalisé
Ligne 8	$i = b$	On rajoute une phrase	La comptine comporte b couplets contenant dans chaque couplet de 1 à b parties du corps de l'alouette. **Invariant Vraie**

On termine enfin en regardant la dernière boucle qui est la ... n-ième.

...

Dernière boucle ($b = n$) : $n - 1$ couplets créés précédemment

	Valeur de i	Nombre de couplets	Invariant
Ligne 3	$i = n$	On rajoute une phrase	On a $n - 1$ couplet finalisé
Ligne 4	$i = n$	On rajoute une phrase contenant 1 partie du corps de l'alouette	On a $n - 1$ couplet finalisé
Lignes 5 à 7	$i = n$	On rajoute une phrase contenant n parties du corps de l'alouette	On a $n - 1$ couplet finalisé
Ligne 8	$i = n$	On rajoute une phrase	La comptine comporte n couplets contenant dans chaque couplet de 1 à n parties du corps de l'alouette. **Invariant Vraie**

La correction partielle est terminée !

c) Terminaison

Dans ce deuxième exemple, il y a deux boucles **POUR**, donc l'algorithme se termine.

d) La complexité

On reprend notre algorithme :

```
1 T ← ['la tête', 'le bec', 'le cou', 'le dos', ...] ;
2 pour i allant de 0 à n − 1 faire
3 |   Chante : ('Alouette, gentille alouette, alouette je te plumerai') ;
4 |   Chante : ('Je te plumerai', T[i], 'je te plumerai', T[i]) ;
5 |   pour j allant de i à 0 faire
6 |   |   Chante : ('et', T[j], 'et', T[j]) ;
7 |   fin pour
8 |   Chante : ('alouette, haaaa') ;
9 fin pour
```

Ligne 1 : 1 opération (on affecte un tableau)
Lignes 2 à 9 : on répète n fois
ligne 3 : 1 opération (on chante)
ligne 4 : 2 opérations (on chante, on affecte récupère la valeur $T[i]$)
lignes 5 à 7 : on répète $i+1$ fois
ligne 6 : 2 opérations (on chante, on affecte récupère la valeur $T[j]$)
ligne 8 : 1 opération (on chante)

Nous obtenons donc :

$$\begin{aligned} NB_{\text{opérations}} &= \underbrace{1}_{\text{ligne 1}} + \underbrace{3+1\times 2+1}_{\text{1ère boucle}} + \underbrace{3+2\times 2+1}_{\text{2ième boucle}} + \underbrace{3+3\times 2+1}_{\text{3ième boucle}} + \ldots \underbrace{3+n\times 2+1}_{n\text{ième boucle}} \\ &= \underbrace{1}_{\text{ligne 1}} + \underbrace{4+1\times 2}_{\text{1ère boucle}} + \underbrace{4+2\times 2}_{\text{2ième boucle}} + \underbrace{4+3\times 2}_{\text{3ième boucle}} + \ldots \underbrace{4+n\times 2}_{n\text{ième boucle}} \\ &= 1 + n\times 4 + (1\times 2 + 2\times 2 + 3\times 2 + \ldots + n\times 2) \\ &= 1 + 4n + 2\times(1+2+3+\ldots+n) \end{aligned}$$

Or nos amis qui ont pris la spécialité mathématiques ont vu ou vont voir cette année que $1+2+3+\ldots+n = \dfrac{n(n+1)}{2}$, donc :

$$\begin{aligned} NB_{\text{opérations}} &= 1 + 4n + 2\times\frac{n(n+1)}{2} \\ &= 1 + 4n + n(n+1) \\ &= n^2 + 5n + 1 \end{aligned}$$

Donc dans cet algorithme, il y a $\boxed{n^2+5n+1}$ opérations si on souhaite avoir n couplets.

Mais maintenant on sait qu'on aurait pu compter plus au moins autrement...

Lorsque la complexité s'exprime sous la forme d'un polynôme du second degré, on dit que

la complexité est quadratique

et on la notera

$$O(n^2)$$

IV. Conclusion

On sait que la fonction $x \mapsto x$ croit beaucoup plus lentement que la fonction $x \mapsto x^2$, donc il vaut mieux avoir une complexité en $O(n)$ que $O(n^2)$.

Mais que veut dire exactement les termes **complexité linéaire** et **complexité quadratique** ?

n	complexité linéaire	complexité quadratique
$n = 10$	il y a de l'ordre d'une dizaine « d'opérations élémentaires » ($n = 10$)	il y a de l'ordre d'une centaine « d'opérations élémentaires » ($n^2 = 100$)
$n = 100$	il y a de l'ordre d'une centaine « d'opérations élémentaires » ($n = 100$)	il y a de l'ordre de dix milliers « d'opérations élémentaires » ($n^2 = 10\ 000$)
$n = 1\ 000$	il y a de l'ordre d'un millier « d'opérations élémentaires » ($n = 1\ 000$)	il y a de l'ordre d'un million « d'opérations élémentaires » ($n^2 = 1\ 000\ 000$)
...	...	...

Donc l'algorithme de **green bottles** fabriquera beaucoup plus efficacement n couplets que l'algorithme **alouette, gentil alouette**.

V. Moralité

L'étude d'un algorithme est longue, compliquée, pénible, avec de forte ressemblance avec des raisonnements mathématiques ... tout pour plaire. Lorsque l'algorithme est relativement simple, on ne s'embête pas à en faire autant. Par contre, en fonction de la difficulté de l'algorithme et de l'utilisation que l'on veut en faire, il peut être intéressant, voir conseiller d'effectuer une étude approfondie de l'algorithme.

Imaginez un algorithme qui permet de trouver une station de radio sur l'autoradio de votre future voiture. S'il trouve la station dans 99% des cas, ça vous convient parfaitement. Tant pis s'il y a un petit risque d'erreur, vous relancez la recherche. Par contre si vous avez un algorithme de freinage de votre future voiture qui fonctionne dans 99% des cas, ..., vu le nombre de freinages que l'on peut faire par jour... cela devient très risqué.

Enfin, imaginez si vous avez un algorithme simple de recherche d'itinéraire à l'aide du système GPS qui fonctionne à tous les coups mais le chemin proposé n'est pas forcément le plus court, vous allez peut-être essayer de vous creuser la tête pour trouver un autre algorithme qui vous permet de trouver à tous les coups l'itinéraire **le plus court**.

VI. Exercices

Exercice 1 (*)
Soit l'algorithme suivant :

```
Entrées : un réel x et un nombre entier n
Sorties : le nombre réel x^n
1 r ← 1 ;
2 pour i allant de 1 à n faire
3 |   r ← r × x ;
4 fin pour
5 retourner r ;
```

1. Donner la spécification de cet algorithme.
2. Donner l'invariant de cet algorithme (correction partielle).
3. Étudier la terminaison de cet algorithme.
4. Étudier la complexité de cet algorithme.

Exercice 2 (*)
Soit l'algorithme suivant :

```
Entrées : un tableau T d'entiers non vide et un nombre entier v
Sorties : un booléen
1 pour i allant de 0 à taille du tableau T - 1 faire
2 |   si T[i] == v alors
3 |   |   trouvee = Vrai ;
4 |   sinon
5 |   |   trouvee = Faux ;
6 |   fin si
7 fin pour
8 retourner trouvee ;
```

1. Donner la spécification de cet algorithme.
2. Donner des tests montrant pour plusieurs raisons que cet algorithme est incorrecte.
3. Proposer un autre algorithme.

Exercice 3 (*)
Voici deux algorithmes qui prétendent effectuer la multiplication de a par b :

Algorithme 1 :

```
   Entrées : deux nombres entiers a et b
   Sorties : le résultat de a × b
1  m ← 0 ;
2  tant que b > 0 faire
3  |  m ← m + a ;
4  |  b ← b − 1 ;
5  fin tq
6  retourner m ;
```

Algorithme 2 :

```
   Entrées : deux nombres entiers a et b
   Sorties : le résultat de a × b
1  i ← 0 ;
2  m ← 0 ;
3  tant que i < b faire
4  |  m ← m + a ;
5  |  i ← i + 1 ;
6  fin tq
7  retourner m ;
```

1. Donner le variant de chacun de ces deux algorithmes.
2. Ces deux algorithmes terminent-ils ?

Exercice 4 (*)
Soient T_1 et T_2 contenant des nombres entiers. Voici deux algorithmes qui calculent la somme de tous les produits de tous nombres de T_1 par tous les nombres de T_2 :

Algorithme 1 :

```
   Entrées : deux tableaux d'entiers T1 et T2
   Sorties : la somme du produit des nombres de T1
             par les nombres de T2
1  somme ← 0 ;
2  pour i allant de 0 à taille de T1 faire
3  |  pour j allant de 0 à taille de T2 faire
4  |  |  somme ← somme + T1[i] × T2[j] ;
5  |  fin pour
6  fin pour
7  retourner somme ;
```

Algorithme 2 :

```
   Entrées : deux tableaux d'entiers T1 et T2
   Sorties : la somme du produit des nombres de T1
             par les nombres de T2
1  somme1 ← 0 ;
2  somme2 ← 0 ;
3  pour i allant de 0 à taille de T1 faire
4  |  somme1 ← somme1 + T1[i] ;
5  fin pour
6  pour j allant de 0 à taille de T2 faire
7  |  somme2 ← somme2 + T2[j] ;
8  fin pour
9  retourner somme1 × somme2 ;
```

1. Expliquer pourquoi ces deux algorithmes produisent le même résultat.
2. Étudier la complexité de ces deux algorithmes en supposant que les deux tableaux soient de même taille n.
   Faire de même avec tableau T_1 est de taille n et que le tableau T_2 est de table m.

Exercice 5 (**)
Si nous avons deux tableaux d'entiers T_1 et T_2 classés dans l'ordre croissant, l'algorithme suivant permet d'afficher toutes les valeurs de T_1 et de T_2 dans le même ordre :

Entrées : deux nombres tableaux d'entiers T_1 et T_2 classés dans l'ordre croissant
Sorties : toutes les valeurs de T_1 et de T_2 dans l'ordre croissant

```
1  i1 ← 0 ;
2  i2 ← 0 ;
3  tant que i1 < taille du tableau T1 ET i2 < taille du tableau T2 faire
4  |  si T1[i1] < T2[i2] alors
5  |  |  afficher T1[i1] ;
6  |  |  i1 ← i1 + 1 ;
7  |  sinon
8  |  |  afficher T2[i2] ;
9  |  |  i2 ← i2 + 1 ;
10 |  fin si
11 fin tq
```

1. Justifier que cet algorithme termine à l'aide de la technique du variant.
2. En supposant que les deux tableaux soient de même taille n, calculer la complexité de cet algorithme.
3. En supposant que le tableau T_1 est de taille n et que le tableau T_2 est de table m recalculer la complexité de cet algorithme.

Correction de l'exercice 1
Soit l'algorithme suivant :

```
Entrées : un réel x et un nombre entier n
Sorties : le nombre réel x^n
1 r ← 1 ;
2 pour i allant de 1 à n faire
3 |   r ← r × x ;
4 fin pour
5 retourner r ;
```

1. Donner la spécification de cet algorithme.

Rôle :	Calculer la puissance d'un nombre « réel ».
Entrées :	Un nombre réel x. Un entier n.
Précondition :	n doit être positif
Sortie :	Un réel
Postcondition :	Renvoie le résultat de x^n

2. Donner l'invariant de cet algorithme (correction partielle).
   r a pour valeur x^i.

3. Étudier la terminaison de cet algorithme.
   C'est une boucle **POUR** donc l'algorithme se termine.

4. Étudier la complexité de cet algorithme.
   Ligne 1 : 1 opération (affectation)
   Lignes 2 à 4 : boucle **POUR**, on répète n fois :
   ligne 3 : 2 opérations (une multiplication puis une affectation)
   Ligne 5 : 1 opération (retourner une valeur)
   On a donc $1 + 2n + 1 = 2n + 2$ opérations. C'est une fonction affine de la variable n.
   La complexité est en $O(n)$, autrement dit elle est **linéaire**.

Correction de l'exercice 2
Soit l'algorithme suivant :

```
Entrées : un tableau T d'entiers non vide et un nombre entier v
Sorties : un booléen
1 pour i allant de 0 à taille du tableau T - 1 faire
2 |   si T[i] == v alors
3 |   |   trouvee = Vrai ;
4 |   sinon
5 |   |   trouvee = Faux ;
6 |   fin si
7 fin pour
8 retourner trouvee ;
```

1. Donner la spécification de cet algorithme.

Rôle :	Trouver une valeur dans un tableau.
Entrées :	Un tableau T Une valeur v.
Préconditions :	T est un tableau non vide d'entiers v est un entier
Sortie :	Un booléen
Postcondition :	Renvoie **Vraie** si la valeur v est bien dans le tableau T

2. Donner des tests montrant pour plusieurs raisons que cet algorithme est incorrecte.
   $T = [12, 15, 14, 16, 12, 11]$
   Pour n'importe quelle valeurs autre que 11, l'algorithme retourne **Faux**.
3. Proposer un autre algorithme.

Entrées : un tableau T d'entiers non vide et un nombre entier v
Sorties : un booléen

```
1 trouvee = Faux ;
2 pour i allant de 0 à taille du tableau T - 1 faire
3 |   si T[i] == v alors
4 |   |   trouvee = Vrai ;
5 |   fin si
6 fin pour
7 retourner trouvee ;
```

Correction de l'exercice 3 (*)
Voici deux algorithmes qui prétendent effectuer la multiplication de a par b :

Algorithme 1 :

Entrées : deux nombres entiers a et b
Sorties : le résultat de $a \times b$

```
1 m ← 0 ;
2 tant que b > 0 faire
3 |  m ← m + a ;
4 |  b ← b − 1 ;
5 fin tq
6 retourner m ;
```

On choisit comme variant b.

- b est impliquée dans la répétitive (lignes 2 et 4)
- b est un entier (précondition)
- b est strictement décroissant (ligne 4)
- b est strictement positif (ligne 2) puis positif (ligne 4)

Donc l'algorithme se termine.

Algorithme 2 :

Entrées : deux nombres entiers a et b
Sorties : le résultat de $a \times b$

```
1 i ← 0 ;
2 m ← 0 ;
3 tant que i < b faire
4 |  m ← m + a ;
5 |  i ← i + 1 ;
6 fin tq
7 retourner m ;
```

i n'est pas un variant car il est strictement croissant (ligne 5).
On choisit comme variant $b - i$.

- $b-i$ est impliquée dans la répétitive (ligne 3) car $i < b$ est équivalent à $b - i > 0$
- $b-i$ est un entier car b entier (précondition) et i entier (ligne 1)
- $b-i$ est strictement décroissant car i strictement croissant (ligne 5)
- $b - i$ est strictement positif (ligne 3) car $i < b$ est équivalent à $b - i > 0$

Donc cet algorithme se termine aussi.