Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :

Eléments	L[0]	L[1]	L[2]	L[3]	...
	↑	↑	↑	↑	↑
Indices	0	1	2	3	...

On peut parcourir cette liste :

Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :

Eléments	L[0]	L[1]	L[2]	L[3]	...
	↑	↑	↑	↑	↑
Indices	0	1	2	3	...

On peut parcourir cette liste :

- Par indice (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :

  ```
  for indice in range(len(L))
  ```

  Il faut alors accéder aux éléments en utilisant leurs indices.

Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :

Eléments	L[0]	L[1]	L[2]	L[3]	...
	↑	↑	↑	↑	↑
Indices	0	1	2	3	...

On peut parcourir cette liste :

- Par indice (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :
  `for indice in range(len(L))`
  Il faut alors accéder aux éléments en utilisant leurs indices.
- Par élément (on se place sur la première ligne du schéma ci-dessus) et on crée une variable qui va parcourir directement la liste des éléments :
  `for element in L`
  La variable de parcours (ici `element`) contient alors directement les éléments).

Algorithme de recherche simple

Un algorithme de recherche simple d'un élément dans une liste consiste :

Algorithme de recherche simple

Un algorithme de recherche simple d'un élément dans une liste consiste :

1. à parcourir les éléments de cette liste

Algorithme de recherche simple

Un algorithme de recherche simple d'un élément dans une liste consiste :

1. à parcourir les éléments de cette liste
2. renvoyer `True` si on trouve cet élément

Algorithme de recherche simple

Un algorithme de recherche simple d'un élément dans une liste consiste :

1. à parcourir les éléments de cette liste
2. renvoyer `True` si on trouve cet élément
3. sinon continuer le parcours et renvoyer `False` si la fin de liste est atteinte

Algorithme de recherche par dichotomie

Lorsqu'une liste est triée, on peut utiliser la recherche par dichotomie :

Algorithme de recherche par dichotomie

Lorsqu'une liste est triée, on peut utiliser la recherche par dichotomie :

1. Partager la liste en deux moitiés

Algorithme de recherche par dichotomie

Lorsqu'une liste est triée, on peut utiliser la recherche par dichotomie :

1. Partager la liste en deux moitiés
2. Comparer l'élément cherché avec celui situé au milieu

Algorithme de recherche par dichotomie

Lorsqu'une liste est triée, on peut utiliser la recherche par dichotomie :

1. Partager la liste en deux moitiés
2. Comparer l'élément cherché avec celui situé au milieu
3. En déduire dans quelle moitié poursuivre la recherche

Algorithme de recherche par dichotomie

Lorsqu'une liste est triée, on peut utiliser la recherche par dichotomie :

1. Partager la liste en deux moitiés
2. Comparer l'élément cherché avec celui situé au milieu
3. En déduire dans quelle moitié poursuivre la recherche
4. S'arrêter lorsque la zone de recherche ne contient plus qu'un élément.

Définition : complexité

La complexité d'un algorithme est une mesure de son efficacité.

Définition : complexité

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

Définition : complexité

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations nécessaire à l'exécution d'un algorithme en fonction de la taille des données

Définition : complexité

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations nécessaire à l'exécution d'un algorithme en fonction de la taille des données
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données

Complexité des algorithmes de recherche

Complexité des algorithmes de recherche

- Le nombre de comparaisons nécessaire à l'algorithme de recherche simple est proportionnel à la taille de la liste.

Complexité des algorithmes de recherche

- Le nombre de comparaisons nécessaire à l'algorithme de recherche simple est proportionnel à la taille de la liste. On dit que cet algorithme a une complexité linéaire.

Complexité des algorithmes de recherche

- Le nombre de comparaisons nécessaire à l'algorithme de recherche simple est proportionnel à la taille de la liste. On dit que cet algorithme a une complexité linéaire.
  L'allure du graphique représentant le temps d'exécution en fonction de la taille des données est une droite.

Complexité des algorithmes de recherche

- Le nombre de comparaisons nécessaire à l'algorithme de recherche simple est proportionnel à la taille de la liste. On dit que cet algorithme a une complexité linéaire.
  L'allure du graphique représentant le temps d'exécution en fonction de la taille des données est une droite.
- Le nombre de comparaisons nécessaire à l'algorithme de recherche dichotomique augmente de 1 lorsque la taille de la liste double.

Complexité des algorithmes de recherche

- Le nombre de comparaisons nécessaire à l'algorithme de recherche simple est proportionnel à la taille de la liste. On dit que cet algorithme a une complexité linéaire.
  L'allure du graphique représentant le temps d'exécution en fonction de la taille des données est une droite.
- Le nombre de comparaisons nécessaire à l'algorithme de recherche dichotomique augmente de 1 lorsque la taille de la liste double. En mathématiques, les fonctions évoluant de cette façon sont appelées logarithmes, l'algorithme de recherche dichotomique a donc une complexité logarithmique

Correction d'un algorithme

On dira qu'un algorithme est correct

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée.

Tests et correction

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct. En effet, ils ne permettent de valider le comportement de l'algorithme que dans quelques cas particuliers et jamais dans le cas général

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

Trouver un invariant de boucle c'est prouver qu'un algorithme fournit la réponse attendue quelque soient les données.

Exemple

On considère la fonction ci-dessous :

```
def compte(elt, liste):
    '''compte le nombre de fois où elt apparaît dans
       liste'''
    compteur=0
    for x in liste:
        if x==elt:
            compteur=compteur+1
    return compteur
```

Exemple

On considère la fonction ci-dessous :

```python
def compte(elt,liste):
    '''compte le nombre de fois où elt apparaît dans
       liste'''
    compteur=0
    for x in liste:
        if x==elt:
            compteur=compteur+1
    return compteur
```

En trouvant un invariant de boucle, montrer qu'à la sortie de la boucle, la variable compteur contient le nombre de fois où `elt` apparaît dans `liste`

Correction de l'exemple

On note la liste $[e_1, e_2, \ldots, e_n]$, et on note `k` le nombre de tours de boucle

Correction de l'exemple

On note la liste $[e_1, e_2, \ldots, e_n]$, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »

Correction de l'exemple

On note la liste $[e_1, e_2, \ldots, e_n]$, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »
est un invariant de boucle.

Correction de l'exemple

On note la liste $[e_1, e_2, \ldots, e_n]$, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »
est un invariant de boucle.

- En entrant dans la boucle (`k`=0) la propriété est vraie car `compteur` vaut 0.

Correction de l'exemple

On note la liste `[`e_1`,`e_2`,...,`e_n`]`, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »
est un invariant de boucle.

- En entrant dans la boucle (`k`$=0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt=`e_{k+1}

Correction de l'exemple

On note la liste $[e_1, e_2, \ldots, e_n]$, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »
est un invariant de boucle.

- En entrant dans la boucle (`k`$=0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`=e_{k+1}

Cette propriété est donc bien un invariant de boucle.

Correction de l'exemple

On note la liste `[`e_1`,`e_2`,...,`e_n`]`, et on note `k` le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les `k` premiers éléments de la liste »
est un invariant de boucle.

- En entrant dans la boucle (`k`$=0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt=`e_{k+1}

Cette propriété est donc bien un invariant de boucle. L'invariant de boucle reste vraie en sortie de boucle ce qui prouve que l'algorithme est correct.

Rappel

On distingue deux types de boucle :

Rappel

On distingue deux types de boucle :

- Les boucles bornées, on connaît leur nombre de répétitions. Ce sont les boucles `for` de Python.

Rappel

On distingue deux types de boucle :

- Les boucles bornées, on connaît leur nombre de répétitions. Ce sont les boucles `for` de Python.
- Les boucles non bornées, qui peuvent se répéter un nombre indéterminé de fois. Ce sont les boucles `while` de Python.

Rappel

On distingue deux types de boucle :

- Les boucles bornées, on connaît leur nombre de répétitions. Ce sont les boucles `for` de Python.
- Les boucles non bornées, qui peuvent se répéter un nombre indéterminé de fois. Ce sont les boucles `while` de Python.

Terminaison d'un algorithme

Une boucle non bornée pouvant se répéter à l'infini, on s'interroge sur le problème de la terminaison d'un programme. C'est à dire qu'on souhaite prouver mathématiquement qu'un programme s'arrête quelques soient les données fournies.

Preuve de la terminaison d'un algorithme

- Pour prouver la terminaison d'un algorithme on utilise la notion de variant de boucle. Il s'agit de mettre en évidence une suite d'entiers naturels strictement décroissante avec le nombre de tours de boucle.

Preuve de la terminaison d'un algorithme

- Pour prouver la terminaison d'un algorithme on utilise la notion de variant de boucle. Il s'agit de mettre en évidence une suite d'entiers naturels strictement décroissante avec le nombre de tours de boucle.
- On montre en mathématique qu'une telle suite ne peut être infinie, traduit en langage informatique cela signifie que la boucle s'arrête forcément.

Exemple

On considère la fonction ci-dessous :

```
def quotient(a,b):
    '''Renvoie le quotient dans la division euclidienne
        de a par b avec a et b deux entiers naturels'''
    q=0
    while a-b>=0:
        a=a-b
        q=q+1
    return q
```

Exemple

On considère la fonction ci-dessous :

```python
def quotient(a,b):
    '''Renvoie le quotient dans la division euclidienne
       de a par b avec a et b deux entiers naturels'''
    q=0
    while a-b>=0:
        a=a-b
        q=q+1
    return q
```

En trouvant un variant de boucle, prouver la terminaison de ce programme.

Correction de l'exemple

Montrer que la suite de valeurs prises par la variable a est une suite d'entiers naturels strictement décroissante.

Correction de l'exemple

Montrer que la suite de valeurs prises par la variable a est une suite d'entiers naturels strictement décroissante.

- La valeur initiale de a est un entier naturel (précondition)

Correction de l'exemple

Montrer que la suite de valeurs prises par la variable $\mathtt{a}$ est une suite d'entiers naturels strictement décroissante.

- La valeur initiale de $\mathtt{a}$ est un entier naturel (précondition)
- A chaque tour de boucle la valeur de $\mathtt{a}$ diminue (de $\mathtt{b}$).

Correction de l'exemple

Montrer que la suite de valeurs prises par la variable `a` est une suite d'entiers naturels strictement décroissante.

- La valeur initiale de `a` est un entier naturel (précondition)
- A chaque tour de boucle la valeur de `a` diminue (de `b`).
- La nouvelle valeur de `a` est `a-b` qui est garantie positive par condition d'entrée dans la boucle

Correction de l'exemple

Montrer que la suite de valeurs prises par la variable `a` est une suite d'entiers naturels strictement décroissante.

- La valeur initiale de `a` est un entier naturel (précondition)
- A chaque tour de boucle la valeur de `a` diminue (de `b`).
- La nouvelle valeur de `a` est `a-b` qui est garantie positive par condition d'entrée dans la boucle

Les trois éléments ci-dessus prouvent que la suite de valeurs prises par la variable `a` est une suite d'entiers naturels strictement décroissante. Cette suite de valeurs est donc finie ce qui prouve la terminaison de cet algorithme.