Algorithmique et Programmation

Algorithmique – Recherche dans une liste

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes

http://www.math-info.univ-paris5.fr/∽bonzon/

Algorithmique

Introduction

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de conditions anormales d'utilisation

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de conditions anormales d'utilisation
- **Réutilisabilité** : aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de conditions anormales d'utilisation
- **Réutilisabilité** : aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu
- **Complexité** : nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu

Algorithme

Algorithme

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

- **Validité** : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de conditions anormales d'utilisation
- **Réutilisabilité** : aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu
- **Complexité** : nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu
- **Efficacité** : aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute

Programme

Programme

Un programme est une suite d'instructions définies dans un langage donné.
Un programme permet de décrire un algorithme.

- Un algorithme exprime la structure logique d'un programme : il est indépendant du langage de programmation
- La traduction de l'algorithme dans un langage de programmation dépend du langage choisi

Pourquoi l'étude des algorithmes ?

- L'étude des algorithmes est fondamentale en informatique.
- L'analyse rigoureuse des algorithmes proposés permet de les **valider**, d'**évaluer leur complexité** et parfois de **justifier de leur optimalité**
- Il faut être capable
  - de s'assurer qu'un programme se termine toujours
  - d'estimer son temps d'exécution pour des valeurs données
  - de déterminer les conditions d'utilisation, de saturation

Calculs élémentaires de complexité

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est “efficace” ou non

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est "efficace" ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est “efficace” ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est “efficace” ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
  - le temps d’execution

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est "efficace" ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
  - le temps d'execution
  - la place utilisée en mémoire

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est "efficace" ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
    - le temps d'execution
    - la place utilisée en mémoire
    - indépendemment de l'implémentation (langage choisi pour programmer, machine utilisée)

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est “efficace” ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
  - le temps d’execution
  - la place utilisée en mémoire
  - indépendemment de l’implémentation (langage choisi pour programmer, machine utilisée)
- On ne veut pas :
  “l’algorithme A, implémenté sur la machine M dans le langage L et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire”

Pourquoi étudier la complexité des algorithmes ?

- Pour savoir si un algorithme est "efficace" ou non
- Pour pouvoir comparer deux algorithmes accomplissant la même tâche
- Pour chaque algorithme, on veut déterminer :
  - le temps d'execution
  - la place utilisée en mémoire
  - indépendemment de l'implémentation (langage choisi pour programmer, machine utilisée)
- On ne veut pas :
  "l'algorithme A, implémenté sur la machine M dans le langage L et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire"
- On veut :
  "Quels que soient l'ordinateur et le langage utilisés, l'algorithme A_1 est **meilleur** *que l'algorithme A_2, pour des données de grandes tailles"*

Qu'est-ce que la complexité d'un algorithme ?

- Il s'agit de caractériser le comportement d'un algorithme sur l'ensemble D_n des données de taille n
- La complexité dépend en général de la taille n des données
- Plusieurs types de complexité :
    - En temps
    - En espace

Qu'est-ce que la complexité d'un algorithme ?

- Il s'agit de caractériser le comportement d'un algorithme sur l'ensemble D_n des données de taille n
- La complexité dépend en général de la taille n des données
- Plusieurs types de complexité :
  - En temps
  - En espace

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :
    - Comparaison entre deux éléments d'une liste

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :
  - Comparaison entre deux éléments d'une liste
  - Affectation d'un élément à une liste

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :
    - Comparaison entre deux éléments d'une liste
    - Affectation d'un élément à une liste
    - ...

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :
  - Comparaison entre deux éléments d'une liste
  - Affectation d'un élément à une liste
  - ...
- Si plusieurs opérations significatives différentes sont choisies, elles doivent être décomptées séparément

Opérations significatives

- Opérations significatives : le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations
- Par exemple, accès à un élément d'une liste :
    - Comparaison entre deux éléments d'une liste
    - Affectation d'un élément à une liste
    - ...
- Si plusieurs opérations significatives différentes sont choisies, elles doivent être décomptées séparément
- En changeant le nombre d'opérations significatives, on varie le degrés de précision de l'analyse

Définitions et notations

- $\text{coût}_A(d)$: complexité de l'algorithme A sur la donnée $d \in D_n$ de taille n

Définitions et notations

- $\text{coût}_A(d)$: complexité de l'algorithme A sur la donnée $d \in D_n$ de taille n
- Complexité au **meilleur des cas** :

$$\text{coût } min_A(n) = min\{\text{coût}_A(d), d \in D_n\}$$

Définitions et notations

- $\text{coût}_A(d)$: complexité de l'algorithme A sur la donnée $d \in D_n$ de taille n
- Complexité au **meilleur des cas** :

$$\text{coût } min_A(n) = min\{\text{coût}_A(d), d \in D_n\}$$

- Complexité au **pire des cas** :

$$\text{coût } max_A(n) = max\{\text{coût}_A(d), d \in D_n\}$$

Définitions et notations

- $\text{coût}_A(d)$: complexité de l'algorithme A sur la donnée $d \in D_n$ de taille n
- Complexité au **meilleur des cas** :

$$\text{coût } min_A(n) = min\{\text{coût}_A(d), d \in D_n\}$$

- Complexité au **pire des cas** :

$$\text{coût } max_A(n) = max\{\text{coût}_A(d), d \in D_n\}$$

- Complexité **moyenne** :

$$\text{coût } moy_A(n) = \sum_{d \in D_n} \text{coût}_A(d) \times p(d)$$

Complexité

- Si deux algorithmes différents effectuent le même travail, il est nécessaire de pouvoir comparer leur complexité

Complexité

- Si deux algorithmes différents effectuent le même travail, il est nécessaire de pouvoir comparer leur complexité
- Il faut alors connaître la rapidité de croissance des fonctions qui mesurent la complexité lorsque la taille des données croît

Complexité

- Si deux algorithmes différents effectuent le même travail, il est nécessaire de pouvoir comparer leur complexité
- Il faut alors connaître la rapidité de croissance des fonctions qui mesurent la complexité lorsque la taille des données croît
- On recherche **l'ordre de grandeur asymptotique**, c'est à dire le coût de l'algorithme à la limite lorsque n devient infini

$$1 \geq \log_2 n \geq n \geq n \log_2 n \geq n^2 \geq n^3 \geq 2^n \dots$$

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
  - constant

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
  - constant
  - logarithmique (Ex : recherche dichotomique)

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
  - constant
  - logarithmique (Ex : recherche dichotomique)
  - linéaire (Ex : recherche séquentielle)

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
  - constant
  - logarithmique (Ex : recherche dichotomique)
  - linéaire (Ex : recherche séquentielle)
  - $n \log n$ (Ex : bons algorithmes de tri)

Complexité

- Plus la taille des données est grande, plus les écarts en temps se creusent
- Les algorithmes utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps
    - constant
    - logarithmique (Ex : recherche dichotomique)
    - linéaire (Ex : recherche séquentielle)
    - $n \log n$ (Ex : bons algorithmes de tri)
- Les algorithmes qui prennent un temps polynomial ne sont utilisables que pour des données de très petite taille

Recherche séquentielle dans une liste non triée

Recherche séquentielle dans une liste non triée

- Soit `liste` une liste non triée de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Recherche séquentielle dans une liste non triée

- Soit `liste` une liste non triée de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste non triée

Parcourir la liste : pour tout $i \in [0, n-1]$, faire :

- Si `liste[i] == e`, retourner `True`
- Sinon, si $i == n-1$, retourner `False`
- Sinon, $i = i + 1$

Recherche séquentielle dans une liste non triée

- Soit `liste` une liste non triée de longueur n, de type `list[elem]`, et e une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste non triée

Parcourir la liste : pour tout $i \in [0, n-1]$, faire :

- Si `liste[i] == e`, retourner `True`
- Sinon, si $i == n-1$, retourner `False`
- Sinon, $i = i + 1$

0	...	i	...	$n-1$
		e ?		

liste[j]$\neq e$ Non testé

Recherche séquentielle dans une liste non triée

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

Recherche séquentielle dans une liste non triée : complexité dans le meilleur cas

```
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`

Recherche séquentielle dans une liste non triée : complexité dans le meilleur cas

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`
- Dépend de `elem`, de `liste` et de `n`

Recherche séquentielle dans une liste non triée : complexité dans le meilleur cas

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`
- Dépend de `elem`, de `liste` et de `n`
- Complexité dans le meilleur des cas :
  Si `liste[0] == elem`, **1 seule comparaison**

Recherche séquentielle dans une liste non triée : complexité dans le pire cas

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`

Recherche séquentielle dans une liste non triée : complexité dans le pire cas

```
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`
- Dépend de `elem`, de `liste` et de `n`

Recherche séquentielle dans une liste non triée : complexité dans le pire cas

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Opérations significatives : `liste[i] == elem` et `n = len(liste)`
- Dépend de `elem`, de `liste` et de `n`
- Complexité dans le **pire des cas** : Si `liste[n-1] == elem`, ou `elem` $\notin$ `liste`, n **comparaisons**

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$
  - Si `elem == liste[i]`, il faut faire $i + 1$ comparaisons

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$
  - Si $\texttt{elem} == \texttt{liste[i]}$, il faut faire $i + 1$ comparaisons
  - Nombre moyen de comparaisons si $\texttt{elem} \in \texttt{liste}$:

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$
  - Si $\texttt{elem} == \texttt{liste[i]}$, il faut faire $i + 1$ comparaisons
  - Nombre moyen de comparaisons si $\texttt{elem} \in \texttt{liste}$:

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Nombre de comparaisons si $\texttt{elem} \notin \texttt{liste}$: n comparaisons.

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$
  - Si `elem == liste[i]`, il faut faire $i + 1$ comparaisons
  - Nombre moyen de comparaisons si $\texttt{elem} \in \texttt{liste}$:

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Nombre de comparaisons si $\texttt{elem} \notin \texttt{liste}$: n comparaisons.
- Complexité moyenne :

$$\text{coût } moy(n) = q\frac{n+1}{2} + (1-q)n$$

Recherche séquentielle dans une liste non triée : complexité moyenne

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1 - q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si $\texttt{elem} \in \texttt{liste}$:
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem} == \texttt{liste[i]}) = \frac{1}{n}$
  - Si $\texttt{elem} == \texttt{liste[i]}$, il faut faire $i + 1$ comparaisons
  - Nombre moyen de comparaisons si $\texttt{elem} \in \texttt{liste}$:

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Nombre de comparaisons si $\texttt{elem} \notin \texttt{liste}$: n comparaisons.
- Complexité moyenne :

$$\text{coût } moy(n) = q\frac{n+1}{2} + (1-q)n$$

- Si $q = \frac{1}{2}$, coût $moy(n) = \frac{3n+1}{4}$. **Complexité de l'ordre de $\frac{3n}{4}$**

Recherche séquentielle dans une liste triée

Liste triée

Liste triée

Une liste de longueur n, de type `list[elem]` est triée si et seulement si :

$$\forall i \in [0, n-2],\ \text{liste[i]} \leq\ \text{liste[i+1]}$$

Si $n = 0$ ou $n = 1$, la liste est triée

Liste triée

Une liste de longueur n, de type `list[elem]` est triée si et seulement si :

$$\forall i \in [0, n-2], \text{ liste[i]} \leq \text{ liste[i+1]}$$

Si $n = 0$ ou $n = 1$, la liste est triée

```
def verif_liste_triee(liste):
    """List --> Bool.
    Vérifie si la liste est triée"""
    trie = True
    i = 0
    n = len(liste)
    while i<(n - 1) and trie :
        if liste[i] > liste[i+1]:
            trie = False
        i = i + 1
    return trie
```

Recherche séquentielle dans une liste triée

- Soit `liste` une liste **triée** de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Recherche séquentielle dans une liste triée

- Soit `liste` une liste **triée** de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste triée

- Si `e > liste[n-1]`, retourner `False`
- Sinon, parcourir la liste : tant que `liste[i] < e`, faire $i = i + 1$
- A la sortie de la boucle,
  - Si `liste[i] == e`, retourner `True`
  - Sinon, retourner `False`

Recherche séquentielle dans une liste triée

- Soit `liste` une liste **triée** de longueur n, de type `list[elem]`, et `e` une élément de type `elem` .
- On cherche s'il existe un indice $i \in [0, n-1]$ tel que `liste[i] == e`

Algorithme de recherche séquentielle dans une liste triée

- Si `e > liste[n-1]`, retourner `False`
- Sinon, parcourir la liste : tant que `liste[i] < e`, faire $i = i + 1$
- A la sortie de la boucle,
  - Si `liste[i] == e`, retourner `True`
  - Sinon, retourner `False`

0	...	i	...	$n-1$
		e ?		

liste[j]$<e$ liste[j]$\geq e$

Recherche séquentielle dans une liste triée

```python
def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
        if liste[i] == elem :
            return True
        else :
            return False
```

Recherche séquentielle dans une liste triée

```
def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
        if liste[i] == elem :
            return True
        else :
            return False
```

Cette méthode est maladroite : efficace si `elem` est présent en début de liste, ou est rapidement plus petit que les éléments de la liste. Autrement, il faut parcourir beaucoup d'éléments !

Recherche séquentielle dans une liste triée

```
def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
        if liste[i] == elem :
            return True
        else :
            return False
```

Cette méthode est maladroite : efficace si `elem` est présent en début de liste, ou est rapidement plus petit que les éléments de la liste. Autrement, il faut parcourir beaucoup d'éléments !
Complexité vue en $O(\frac{n}{2})$ (vu en TD)

Recherche dichotomique

Recherche dichotomique

- Il est possible d'être beaucoup plus efficace si le vecteur est trié
- Méthode dichotomique :
  - Comparer l'élément e à une valeur située au milieu du vecteur
  - Si cette valeur est différente de e, continuer la recherche sur le demi-vecteur susceptible de contenir e

Explication de l'algorithme

Partages successifs sur un vecteur **trié**

- Situation générale

0	...	inf	...	med	...	sup	...	$n-1$

liste[j]$<e$ — A explorer — liste[j]$\geq e$

$$0 \leq inf \leq sup \leq n-1$$

- 2 cas possibles :
  - $inf \leq sup$: on pose $med = \lfloor \frac{(inf+sup)}{2} \rfloor$
    - `e == liste[med]`, retourner `True`
    - `e < liste[med]`, $sup = med - 1$
    - `e > liste[med]`, $inf = med + 1$
  - $inf > sup$: retourner `False`
- Conditions initiales : $inf = 0$, $sup = n - 1$.

Recherche dichotomique dans une liste triée

```python
def recherche_dichotomique(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    appartient = False
    inf, sup = 0, len(liste) - 1

    while inf <= sup and not(appartient) :
        med = (inf + sup)//2
        if liste[med] == elem :
            appartient = True
        elif liste[med] > elem:
            sup = med - 1
        else :
            inf = med + 1
    return appartient
```

Recherche dichotomique dans une liste triée

```python
def recherche_dichotomique(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    appartient = False
    inf, sup = 0, len(liste) - 1

    while inf <= sup and not(appartient) :
        med = (inf + sup)//2
        if liste[med] == elem :
            appartient = True
        elif liste[med] > elem:
            sup = med - 1
        else :
            inf = med + 1
    return appartient
```

Complexité en $O(\log n)$

Pour conclure

Résumé du cours

Aujourd'hui, on a vu

- L'importance de l'algorithmique
- Quelques éléments du calcul de la complexité
- Les algorithmes classiques de recherche d'un élément dans une liste, triée ou non