Objectifs
- connaître les algorithmes de recherche séquentielle et dichotomique ;
- savoir écrire ces algorithmes en itératif et en récursif ;
- s'initier au calcul du coût d'un algorithme.

1 Spécification du problème

1.1 Introduction

À de très nombreuses occasions, un programmeur est amené à soumettre l'exécution d'un calcul ou d'une instruction à la condition qu'une certaine valeur x est ou n'est pas dans une collection ℓ. En Python, ce programmeur écrit un code qui peut prendre la forme :

```
...
if x in l:
...
```

Parfois, ce programmeur ne peut pas se contenter d'une réponse booléenne : il lui faut en plus savoir où se trouve cet élément. Dans son code Python, il écrit alors une expression de la forme :

```
...
l.index(x)
...
```

Étant donnés une structure de données ℓ (la collection de données dans laquelle on cherche) et une valeur x (l'élément que l'on cherche), plusieurs types de problèmes peuvent être posés :

- l'élément est-il ou non dans la collection ? autrement dit $x \in \ell$? si
- l'élément est dans la collection, où peut-on le trouver ? autrement dit, pour
- quel indice i a-t-on $\ell[i] = x$?

  Cette dernière question pouvant être déclinée de plusieurs façons :

  – quel est le plus petit indice i tel que $\ell[i] = x$?
  – quel est le plus grand ?
  – quels sont tous les indices ?

Dans ce qui suit, nous nous contenterons d'indiquer comment répondre au premier problème, c'est-à-dire l'appartenance ou non d'un élément à une collection. Autrement dit nous nous intéresserons plus à l'opérateur `in` qu'à la méthode `index`, mais il est très facile d'adapter les solutions du premier problème au second (en tout cas pour ses deux premières déclinaisons : plus petit indice, plus grand indice).

Nous allons traiter un problème un peu différent que celui que traite l'opérateur `in` : nous nous donnerons non seulement l'élément à chercher et la collection dans laquelle le chercher, mais aussi deux indices donnant les extrémités de l'intervalle de recherche.

Dit autrement, la question que nous allons résoudre est

$$x \in \ell[a : b] \,?$$

où $\ell[a : b]$ désigne la tranche des éléments de ℓ dont les indices sont compris entre a (inclus) et b (exclu), en suivant les notations usuelles de Python.

1.2 Fonction de comparaison

De plus, si ℓ est une liste dont les éléments sont de type `A`, nous utiliserons pour les comparer une fonction `comp` d'entête :

```
def comp(x: A, y: A) -> int:
    """Compare x et y.

    Renvoie :
      - -1 si x < y
```

```
      - 0 si x == y
      - 1 si x > y
    """
```

1.3 Spécification des algorithmes de comparaison

La spécification des algorithmes de recherche est donc :

```
def recherche(elt: A, liste: list[A],
              a:int = 0, b: int = None,
              comp: Callable[[A, A], int]=compare) -> bool:
    """Renvoie :
      - True si elt est dans liste[a:b] (recherche fructueuse)
      - False sinon (recherche infructueuse)

    Precondition: les éléments de liste et elt sont comparables
           0 <= a <= b <= len(liste)

    Exemples:
    $$$ l = [2 * k for k in range (100)]
    $$$ recherche(2 * random.randrange(100), l, 0, 100)
    True
    $$$ recherche(2 * random.randrange(-50, 100) + 1, l, 0, 100)
    False
    """
```

Dans la suite nous déclarerons *fructueuse* toute recherche d'élément qui réussit (sortie *Vrai* dans la spécification ci-dessus). Dans le cas contraire nous dirons qu'elle *échoue* ou qu'elle est *non fructueuse*.

2 Recherches dans une structure séquentielle non triée

> L'idée de la recherche séquentielle est de parcourir dans l'ordre des indices les éléments de la tranche jusqu'à trouver l'élément recherché ou atteindre l'extrémité de la tranche.

2.1 Implantation récursive

```
def recherche_seq_rec(elt: A, liste: list[A],
                      a: int = 0, b: int = None,
                      comp: Callable[[A, A], int] = compare) -> bool:
    if b is None:
        b = len(liste)
    if b - a > 0:
        return comp(elt, liste[a]) == 0 or \
            recherche_seq_rec(elt, liste, a+1, b, comp)
    else:
        return False
```

- cet algorithme récursif se termine : Par exemple $v(a, b) = b - a$ est un variant ;
- cet algorithme est correct. Pour le démontrer, nous pouvons effectuer une démonstration par récurrence sur la longueur $n = b - a$ de la tranche.

1. Pour les listes de longueur nulle, la fonction renvoie False, ce qui est correct.
2. Supposons que la fonction soit correcte pour toutes les tranches de liste de longueur $< n$, et considérons une tranche de longueur n.
   - si l'élément est égal au premier élément de la tranche, alors la fonction renvoie `True`, ce qui est correct ;
   - sinon, la recherche s'effectue dans une tranche de longueur $n-1$ pour laquelle elle renvoie un résultat correct par hypothèse de récurrence.

2.2 Implantation itérative

```
def recherche_seq(elt: A, liste: list[A],
                  a: int = 0, b: int = None,
```

```
                comp: Callable[[A, A], int] = compare) -> bool:
    if b is None:
        b = len(liste)
    trouve = False
    i = a
    while not trouve and i < b:
        if comp(elt, liste[i]) == 0:
            trouve = True
        i += 1
    return trouve
```

Pour démontrer la terminaison de la recherche séquentielle itérative, on peut utiliser un *variant de boucle*. Comme les variants d'algorithmes récursifs, les variants de boucle sont la fonction $v(x1, x2, \dots, xp)$ des variables utilisées lors de l'itération, de telle sorte qu'elles définissent des suites d'entiers naturels strictement décroissantes.

Par exemple, la fonction v définie par $v(i) = b - i$ est un variant de boucle :

- elle est à valeur entière ;
- la condition de boucle nous assure que i < b en début de boucle, et donc que $v(i) > 0$ en début de boucle ;
- on a $v(i+1) = b - (i+1) = b - i - 1 = v(i) - 1$ et donc
  - $v(i) \geq 0$ en fin de boucle et
  - v est strictement décroissante.

Comme toute suite strictement décroissante d'entiers est finie, on en déduit que cet algorithme se termine.

La correction repose sur une *propriété invariante*. Ici on peut choisir

'(elt $\notin$ liste[0:i]) ET (trouve vaut Faux OU liste[i-1] = elt)'

1. Cette propriété est vraie avant la boucle car la tranche liste[:i] est vide et trouve vaut Faux.
2. De plus, si elle est vraie avant le corps de la boucle, alors elle est vraie après le corps ...
3. La négation de la condition et cette propriété nous assure que :
   - Soit trouve est Faux et aucun élément de la liste n'est égal à elt ;
   - Soit trouve est Vrai et i-1 est l'indice d'un élément égal à elt.

2.3 Coût de la recherche séquentielle

Pour comparer deux algorithmes, on peut comparer leur coût. Ici on suppose que les algorithmes de recherche vont opérer en comparant deux éléments à la fois et on décide donc de définir le coût d'un algorithme de recherche comme :

> le nombre d'appel à la fonction `comp` pour fournir la réponse, en fonction de la longueur n de la tranche.

Ce nombre ne peut toutefois pas toujours être calculé : en effet, le nombre de comparaisons dépend :

- de la *longueur* de la tranche et
- des *éléments* contenus dans la tranche.

On se contente donc d'encadrer ce coût en trouvant pour une tranche de longueur n :

- un meilleur des cas : il s'agit d'une liste pour laquelle `liste[a]` est l'élément recherché. Dans ce cas on effectue une seule comparaison ;
- des pires des cas : il s'agit de toutes les listes ne contenant pas l'élément recherché. Dans ce cas on effectue n comparaisons (toutes évaluées à `False`)

et donc

$$1 \leq c(n) \leq n$$

3 Recherches dans une structure triée

Sans renseignement sur les propriétés de la liste et de ses éléments, le coût ne peut pas être amélioré. Dans cette partie nous allons améliorer le coût dans le cas où la liste est **triée**.

Il faut avant tout définir ce que l'on entend par liste triée :

> Définition
> Une liste ℓ est dite triée par rapport à la fonction `comp` si
> $\forall i.0 \leq i \wedge i < len(\ell) - 1 \Rightarrow \text{comp}(\ell[i], \ell[i+1]) <= 0$

Cela signifie qu'une liste triée l'est dans l'ordre **croissant** et par rapport à la fonction **comp**.

Par exemple, considérons la liste de chaînes de caractères :

```
>>> liste = ["z", "un" ,"deux", "quatre"]
```

Cette liste n'est pas triée pour l'ordre lexicographique des caractères. En effet, dans cet ordre on a `"z"` qui est après `"un"`. Par contre elle l'est si on compare nos chaînes de caractères par longueur.

Le prédicat `est_triee` peut donc s'écrire ainsi :

```
def est_triee(liste: list[A], comp: Callable[[A, A], int]=compare) -> bool:
    for i in range(len(liste) - 1):
        if comp(liste[i], liste[i+1]) > 0:
            return False
    return True
```

3.1 la recherche dichotomique

L'**idée** de la recherche dichotomique est de profiter de l'ordre des valeurs dans la liste. Pour une tranche `[a:b]` donnée dans laquelle on recherche `elt`, on cherche l'indice m du milieu de cette tranche.

- Si `elt > liste[m]`, alors on peut poursuivre la recherche dans la tranche [m+1:b]
- Sinon la recherche se poursuit dans la tranche [a:m+1]

Pour que l'algorithme se termine, il faut que la longueur des tranches décroît strictement. C'est le cas lorsque la tranche contient au moins deux éléments, c'est-à-dire lorsque $a < b - 1$.

Lorsque cette condition n'est plus remplie, la tranche contient zéro ou un élément et il est facile de déterminer la réponse.

3.1.1 Implantation récursive

```
def recherche_dicho_rec(elt: A, liste: list[A],
                a:int = 0, b: int = None,
                comp: Callable[[A, A], int]=compare) -> bool:
    """
    precondition: liste est triée selon `comp`
    """
    if b is None:
        b = len(liste)
    d = a
    f = b - 1
    if d < f:
        m = (d + f) // 2
        if comp(elt, liste[m]) > 0:
            res = recherche_dicho_rec(elt, liste, m+1, f+1, compare)
        else:
            res = recherche_dicho_rec(elt, liste, d, m+1, compare)
    else:
        res = (a < b) and comp(elt, liste[d]) == 0
    return res
```

3.1.2 Implantation itérative

```
def recherche_dicho(elt: A, liste: list[A],
                    a: int = 0, b: int = None,
                    comp: Callable[[A, A], int] = compare,
                    **kwargs) -> bool:
    """
    precondition: liste est triée selon `comp`
    """
```

```
    if b is None:
        b = len(liste)
    d = a
    f = b - 1
    trace = kwargs.get('trace', False)
    if trace:
        print(f"d={d}, f={f}")
    while d < f:
        m = (d + f) // 2
        if comp(elt, liste[m]) == 1:
            d = m + 1
        else:
            f = m
        if trace:
            print(f"d={d}, f={f}")
    return (a < b) and comp(elt, liste[d]) == 0
```

3.1.3 Terminaison et correction

- Dans les deux implantations, on peut choisir comme variant la longueur de la tranche dans laquelle s'effectue la recherche.
- Pour la correction, on peut remarquer que la propriété

  elt ∈ liste ⇒ elt ∈ liste[a:b]

  est toujours vraie dans le cas récursif, et

  elt ∈ liste ⇒ elt ∈ liste[d:f+1]

  est toujours vraie dans le cas itératif.

3.1.4 Coût de la recherche dichotomique

Pour évaluer le coût de notre algorithme de recherche dichotomique, on peut se placer tout d'abord dans le cas où la longueur n de la tranche est une puissance de 2 :

$$n = 2^p$$

Notons $c_{\text{dico}}(n)$ le nombre de comparaisons d'éléments de la liste nécessaires à l'algorithme pour fournir sa réponse sur une liste de longueur n.

On a

$$\left\{ \begin{array}{rcl} c_{\text{dico}}(n) & = & c_{\text{dico}}(n \div 2) + 1 \text{ si } n > 0 \\ c(0) & = & 0 \end{array} \right.$$

En fonction de p, on a :

$$\left\{ \begin{array}{rcl} c'_{\text{dico}}(p) & = & c'_{\text{dico}}(p-1) + 1 \\ c'(0) & = & c(1) = 1 \end{array} \right.$$

Il s'agit d'une suite arithmétique de raison 1 et donc $c'(p) = p + 1$

On en déduit que

$$c(n) = p + 1 = \log_2(n) + 1$$

Dans le cas où n est quelconque, on peut supposer que le coût est une fonction croissante et donc :

$$2^p \leq n < 2^{p+1} \Rightarrow p + 1 \leq c(n) < p + 2$$

Et donc $c(n) = p + 1$. Comme $p \leq \log_2(n) < p + 1$, on en déduit que $p = \lfloor \log_2(n) \rfloor$

et donc finalement

$$c(n) = \lfloor \log_2(n) \rfloor + 1$$