TP 4 – Recherche dichotomique

Objectifs

- Comprendre le principe d'une recherche dichotomique.
- Savoir maintenir une tranche de recherche dans une liste triée.
- Écrire une boucle `while` avec une condition d'arrêt correcte.
- Comparer une recherche naïve et une recherche dichotomique.
- Découvrir un autre exemple de division par deux : l'exponentiation rapide.

Travail en classe

Exercice 1 – Recherche dichotomique dans une liste triée

On dispose d'une liste Python `lst` *triée* dans l'ordre croissant, par exemple :

```
lst = [2, 5, 7, 9, 12, 15, 18, 21, 26, 31]
```

On cherche à déterminer si un élément `elt` apparaît dans cette liste. Une *recherche naïve* consiste à parcourir les éléments de la liste un par un. Mais, puisque la liste est *triée*, on peut faire mieux.

Question 1. On cherche l'élément `18` dans la liste précédente. À la main, appliquer le principe de la recherche dichotomique :

- comparer `elt` avec l'élément situé au milieu de la zone de recherche ;
- si l'élément du milieu est trop petit, chercher à droite ;
- s'il est trop grand, chercher à gauche ;
- s'il est égal à `elt`, la recherche est terminée.

Indiquer, à chaque étape, la zone de recherche et l'indice examiné.

Question 2. Même question pour l'élément `10`. Que se passe-t-il lorsque la zone de recherche devient vide ?

Question 3. On représente la zone de recherche par deux indices `gauche` et `droite`. À tout instant, on cherche l'élément dans la tranche `lst[gauche:droite]`. L'indice `gauche` est inclus, tandis que l'indice `droite` est exclu. Au départ, quelles valeurs faut-il donner à `gauche` et `droite` ?

Question 4. On suppose que la tranche de recherche est non vide. On note :

```
milieu = (gauche + droite) // 2
```

Expliquer comment mettre à jour `gauche` ou `droite` dans chacun des cas suivants :

- `lst[milieu] == elt` ;
- `lst[milieu] < elt` ;
- `lst[milieu] > elt`.

On prendra soin de justifier pourquoi l'indice `milieu` peut être éliminé de la suite de la recherche lorsque `lst[milieu] != elt`.

Question 5. Compléter la fonction suivante.

```
def present_dicho(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""
    gauche = ...
    droite = ...

    while ...:
        milieu = ...

        if ...:
            return True
        elif ...:
            gauche = ...
        else:
            droite = ...

    return False
```

Question 6. Tester la fonction sur plusieurs cas.

```
assert present_dicho([2, 5, 7, 9, 12], 2)
assert present_dicho([2, 5, 7, 9, 12], 12)
assert present_dicho([2, 5, 7, 9, 12], 7)
assert not present_dicho([2, 5, 7, 9, 12], 8)
assert not present_dicho([], 3)
assert present_dicho([4], 4)
assert not present_dicho([4], 5)
```

Exercice 2 – Comparer recherche naïve et recherche dichotomique

On souhaite comparer expérimentalement une recherche naïve et une recherche dichotomique.

Question 1. Écrire une fonction `present_naif(lst: list, elt: int) -> bool` qui renvoie `True` si `elt` apparaît dans `lst`, et `False` sinon, en parcourant la liste de gauche à droite. On n'utilisera pas l'opérateur `in`.

Question 2. Écrire une fonction `nb_comparaisons_naif(lst: list, elt: int) -> int` qui renvoie le nombre de valeurs de la liste comparées à `elt` lors de la recherche naïve.
Par exemple, si `lst = [2, 5, 7, 9, 12]`, alors la recherche de `7` examine trois valeurs, tandis que la recherche de `13` examine les cinq valeurs de la liste.

Question 3. Écrire une fonction `nb_comparaisons_dicho(lst: list, elt: int) -> int` qui renvoie le nombre de valeurs de la liste examinées lors de la recherche dichotomique. Autrement dit, on compte le nombre d'indices `milieu` calculés pendant la recherche. On compte une valeur examinée chaque fois qu'un nouvel indice `milieu` est calculé, même si le programme teste ensuite plusieurs conditions portant sur `lst[milieu]`.
Par exemple, si `lst = [2, 5, 7, 9, 12]`, la recherche dichotomique de `7` examine une seule valeur, tandis que celle de `13` en examine deux.

Question 4. Pour plusieurs valeurs de n, on considère la liste `list(range(n))`. Compléter expérimentalement le tableau suivant en recherchant un élément absent, par exemple `n`, puis commenter les résultats obtenus :

n	10	100	1000	10000
recherche naïve	...	...	...	...
recherche dichotomique	...	...	...	...

Travail à la maison

Exercice 3 – Recherche dichotomique récursive

On souhaite réécrire la recherche dichotomique de manière récursive.

Question 1. Une première possibilité consiste à utiliser le slicing. À chaque appel récursif, on construit la moitié de la liste dans laquelle la recherche doit se poursuivre. Compléter la fonction suivante.

```
def present_dicho_rec_slice(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""
    if lst == []:
        return False

    milieu = len(lst) // 2

    if ...:
        return True
    elif ...:
        return present_dicho_rec_slice(..., elt)
    else:
        return present_dicho_rec_slice(..., elt)
```

Question 2. Tester cette fonction sur les mêmes exemples que la fonction `present_dicho`.

Question 3. Quel est l'inconvénient du slicing dans cette version ?

Question 4. Pour éviter de construire de nouvelles listes, on utilise une *fonction récursive auxiliaire* qui reçoit les bornes de la tranche de recherche. Comme dans la version impérative, on cherche l'élément dans la tranche `lst[gauche:droite]` ; l'indice `gauche` est inclus, l'indice `droite` est exclu. Compléter la fonction suivante.

```
def present_dicho_rec(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""

    def aux(gauche: int, droite: int) -> bool:
        """Recherche elt dans la tranche lst[gauche:droite]."""
        if ...:
            return False

        milieu = ...

        if ...:
            return True
        elif ...:
            return aux(..., ...)
        else:
            return aux(..., ...)

    return aux(0, len(lst))
```

Question 5. Comparer cette version avec la version impérative.

Pour aller plus loin

Exercice 4 – Exponentiation rapide récursive

On cherche à calculer efficacement x^n, où x est un entier et n un entier naturel. La méthode naïve multiplie x par lui-même $n-1$ fois si n est non nul. Elle effectue donc un nombre de multiplications proportionnel à n. On peut faire mieux en utilisant le fait que $x^0 = 1$ et que :

$$\forall n \in \mathbb{N}^* \qquad x^n = \begin{cases} \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair} \\ x \times \left(x^{(n-1)/2}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

Dans le programme Python, ces deux exposants seront obtenus par `n // 2`.

Question 1. Expliquer pourquoi cette méthode divise la taille du problème par deux à chaque appel récursif.

Question 2. Compléter la fonction suivante.

```
def puissance_rapide_rec(x: int, n: int) -> int:
    """Renvoie x puissance n, où n est un entier naturel."""
    if n == 0:
        return ...

    y = puissance_rapide_rec(x, n // 2)

    if n % 2 == 0:
        return ...
    else:
        return ...
```

Question 3. Tester la fonction.

```
assert puissance_rapide_rec(2, 0) == 1
assert puissance_rapide_rec(2, 1) == 2
assert puissance_rapide_rec(2, 10) == 1024
assert puissance_rapide_rec(3, 5) == 243
assert puissance_rapide_rec(5, 3) == 125
```

Question 4. Comparer le nombre d'appels récursifs nécessaires avec celui de la méthode naïve.

Exercice 5 – Exponentiation rapide impérative

Cet exercice est plus difficile que le précédent.
L'exponentiation rapide peut aussi s'écrire sans récursion. L'idée consiste à maintenir trois variables : `res`, qui contient le résultat partiel ; `base`, qui contient une puissance de x ; `exp`, qui contient la partie de l'exposant restant à traiter. Au départ `res = 1`, `base = x`, `exp = n`. À chaque étape :

- si `exp` est impair, on multiplie `res` par `base` ;
- on remplace `base` par `base * base` ;
- on remplace `exp` par `exp // 2`.

Question 1. Construire le tableau des valeurs successives de `res`, `base`, `exp` qui permettent de calculer 3^{13}.

Question 2. Compléter la fonction suivante.

```
def puissance_rapide(x: int, n: int) -> int:
    """Renvoie x puissance n, où n est un entier naturel."""
    res = ...
    base = ...
    exp = ...

    while ...:
        if ...:
            res = ...
        base = ...
        exp = ...

    return res
```

Question 3. Tester la fonction sur les mêmes exemples que la version récursive.

Question 4. Pendant l'exécution de la boucle, quelle relation lie les variables `res`, `base`, `exp`, ainsi que les valeurs initiales de `x` et `n` ?

Corrigés des exercices

Corrigé — Exercice 1 — Recherche dichotomique dans une liste triée

Question 1. Les étapes de la recherche de `18` sont les suivantes :

`gauche`	`droite`	`milieu`	`lst[milieu]`
0	10	5	15
6	10	8	26
6	8	7	21
6	7	6	18

La valeur est trouvée à l'indice 6.

Question 2. Pour rechercher `10`, on obtient :

`gauche`	`droite`	`milieu`	`lst[milieu]`
0	10	5	15
0	5	2	7
3	5	4	12
3	4	3	9

On obtient ensuite `gauche = droite = 4`. La tranche de recherche est vide : la valeur `10` n'appartient pas à la liste.

Question 3. Au départ, la zone de recherche est la liste entière. Il faut donc initialiser les bornes avec :

```
gauche = 0
droite = len(lst)
```

Question 4. Trois cas se présentent :

- si `lst[milieu] == elt`, la fonction peut immédiatement renvoyer `True` ;
- si `lst[milieu] < elt`, la recherche se poursuit à droite et on pose `gauche = milieu + 1` ;
- si `lst[milieu] > elt`, la recherche se poursuit à gauche et on pose `droite = milieu`.

Lorsque `lst[milieu] != elt`, l'indice `milieu` peut être éliminé : la liste étant triée, la comparaison permet de savoir dans quelle moitié la valeur pourrait encore se trouver.

Question 5.

```
def present_dicho(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""
    gauche = 0
    droite = len(lst)

    while gauche < droite:
        milieu = (gauche + droite) // 2

        if lst[milieu] == elt:
            return True
        elif lst[milieu] < elt:
            gauche = milieu + 1
        else:
            droite = milieu

    return False
```

Question 6. Les tests proposés s'exécutent sans déclencher d'erreur. On peut les compléter avec des listes contenant des valeurs répétées :

```
assert present_dicho([1, 1, 1, 2, 2], 1)
assert present_dicho([1, 1, 1, 2, 2], 2)
assert not present_dicho([1, 1, 1, 2, 2], 3)
```

Corrigé — Exercice 2 — Comparer recherche naïve et recherche dichotomique

Question 1.

```
def present_naif(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à lst."""
    for valeur in lst:
        if valeur == elt:
            return True
    return False
```

Question 2. La variable `nb` compte les valeurs successivement examinées.

```
def nb_comparaisons_naif(lst: list, elt: int) -> int:
    """Compte les valeurs examinées par la recherche naïve."""
```

```
    nb = 0
    for valeur in lst:
        nb = nb + 1
        if valeur == elt:
            return nb
    return nb
```

Ainsi :

```
lst = [2, 5, 7, 9, 12]
assert nb_comparaisons_naif(lst, 7) == 3
assert nb_comparaisons_naif(lst, 13) == 5
```

Question 3. On augmente cette fois le compteur chaque fois qu'un nouvel indice `milieu` est calculé.

```
def nb_comparaisons_dicho(lst: list, elt: int) -> int:
    """Compte les valeurs examinées par recherche dichotomique."""
    gauche = 0
    droite = len(lst)
    nb = 0

    while gauche < droite:
        milieu = (gauche + droite) // 2
        nb = nb + 1

        if lst[milieu] == elt:
            return nb
        elif lst[milieu] < elt:
            gauche = milieu + 1
        else:
            droite = milieu

    return nb
```

Avec la même liste :

```
assert nb_comparaisons_dicho(lst, 7) == 1
assert nb_comparaisons_dicho(lst, 13) == 2
```

Question 4. On obtient expérimentalement :

n	10	100	1000	10000
recherche naïve	10	100	1000	10000
recherche dichotomique	3	6	9	13

Le nombre de valeurs examinées par la recherche naïve est, dans le pire des cas, proportionnel à la taille n de la liste. Pour la recherche dichotomique, il est de l'ordre de $\log_2(n)$. La recherche dichotomique devient donc nettement plus avantageuse lorsque la taille de la liste augmente.

Corrigé – Exercice 3 – Recherche dichotomique récursive

Question 1.

```
def present_dicho_rec_slice(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""
    if lst == []:
        return False

    milieu = len(lst) // 2

    if lst[milieu] == elt:
        return True
    elif lst[milieu] < elt:
        return present_dicho_rec_slice(lst[milieu + 1:], elt)
    else:
        return present_dicho_rec_slice(lst[:milieu], elt)
```

Question 2. On peut reprendre exactement les tests de la version impérative.

```
assert present_dicho_rec_slice([2, 5, 7, 9, 12], 2)
assert present_dicho_rec_slice([2, 5, 7, 9, 12], 12)
assert present_dicho_rec_slice([2, 5, 7, 9, 12], 7)
assert not present_dicho_rec_slice([2, 5, 7, 9, 12], 8)
assert not present_dicho_rec_slice([], 3)
```

Question 3. Chaque opération de slicing construit une nouvelle liste et recopie les éléments de la tranche extraite. Cette version utilise donc davantage de temps et de mémoire que la version qui ne manipule que des indices.

Question 4.

```
def present_dicho_rec(lst: list, elt: int) -> bool:
    """Renvoie True si elt appartient à la liste triée lst."""

    def aux(gauche: int, droite: int) -> bool:
        """Recherche elt dans la tranche lst[gauche:droite]."""
        if gauche >= droite:
            return False

        milieu = (gauche + droite) // 2

        if lst[milieu] == elt:
            return True
        elif lst[milieu] < elt:
            return aux(milieu + 1, droite)
        else:
            return aux(gauche, milieu)

    return aux(0, len(lst))
```

Question 5. La version récursive avec fonction auxiliaire et la version impérative examinent les mêmes indices. Elles effectuent donc un nombre comparable d'opérations. La version récursive utilise toutefois la pile d'appels, tandis que la version impérative ne nécessite que les variables `gauche`, `droite` et `milieu`.

Corrigé — Exercice 4 — Exponentiation rapide récursive

Question 1. À chaque appel récursif, l'exposant `n` est remplacé par `n // 2`. La taille de l'exposant restant à traiter est donc, à une unité près, divisée par deux.

Question 2.

```
def puissance_rapide_rec(x: int, n: int) -> int:
    """Renvoie x puissance n, où n est un entier naturel."""
    assert n >= 0
    if n == 0:
        return 1

    y = puissance_rapide_rec(x, n // 2)
    if n % 2 == 0:
        return y * y
    else:
        return x * y * y
```

Le résultat de l'appel récursif est mémorisé dans `y` afin de ne pas effectuer deux fois le même calcul.

Question 3. Les tests proposés permettent de vérifier le cas de base ainsi que les cas où l'exposant est pair ou impair. On peut les compléter par :

```
for x in range(-5, 6):
    for n in range(10):
        assert puissance_rapide_rec(x, n) == x**n
```

Question 4. Avec la méthode naïve, l'exposant diminue d'une unité à chaque appel : le nombre d'appels est proportionnel à n. Avec l'exponentiation rapide, il est divisé par deux : le nombre d'appels est de l'ordre de $\log_2(n)$.

Corrigé — Exercice 5 — Exponentiation rapide impérative

Question 1. Pour calculer 3^{13}, les valeurs prises par les variables à la fin de chaque passage dans la boucle sont les suivantes.

res	base	exp
1	3	13
3	9	6
3	81	3
243	6561	1
1594323	43046721	0

La première ligne donne les valeurs avant la boucle ; les suivantes donnent les valeurs à la fin de chaque passage.

Question 2.

```
def puissance_rapide(x: int, n: int) -> int:
    """Renvoie x puissance n, où n est un entier naturel."""
    assert n >= 0
    res = 1
    base = x
    exp = n
```

```
    while exp > 0:
        if exp % 2 == 1:
            res = res * base
        base = base * base
        exp = exp // 2

    return res
```

Question 3.

```
assert puissance_rapide(2, 0) == 1
assert puissance_rapide(2, 1) == 2
assert puissance_rapide(2, 10) == 1024
assert puissance_rapide(3, 5) == 243
assert puissance_rapide(5, 3) == 125
```

Question 4. À tout moment pendant l'exécution de la boucle, les variables `res`, `base` et `exp` vérifient la relation suivante :

$$\texttt{res} \times \texttt{base}^{\texttt{exp}} = x^n$$

Autrement dit, `res` contient la partie du résultat déjà calculée, tandis que `base**exp` représente la partie qu'il reste à calculer.
Au début, cette égalité est vérifiée puisque `res = 1`, `base = x` et `exp = n`. Les instructions de la boucle modifient les trois variables sans changer la valeur du produit.
À la fin, `exp` vaut 0. Comme toute puissance d'exposant nul vaut 1, la relation précédente devient `res = x**n`. La variable `res` contient donc le résultat attendu.