TP2__Python

September 11, 2017

1 Table of Contents

1 TP 2 - Programmation pour la préparation à l'agrégation maths option info
2 Listes
2.1 Exercice 1 : taille
2.2 Exercice 2 : concat
2.3 Exercice 3 : appartient
2.4 Exercice 4 : miroir
2.5 Exercice 5 : alterne
2.6 Exercice 6 : nb_occurrences
2.7 Exercice 7 : pairs
2.8 Exercice 8 : range
2.9 Exercice 9 : premiers
3 Quelques tris par comparaison
3.1 Exercice 10 : Tri insertion
3.2 Exercice 11 : Tri insertion générique
3.3 Exercice 12 : Tri selection
3.4 Exercices 13, 14, 15 : Tri fusion
3.5 Comparaisons
4 Listes : l'ordre supérieur
4.1 Exercice 16 : applique
4.2 Exercice 17
4.3 Exercice 18 : itere
4.4 Exercice 19
4.5 Exercice 20 : qqsoit et ilexiste
4.6 Exercice 21 : appartient version 2
4.7 Exercice 22 : filtre
4.8 Exercice 23
4.9 Exercice 24 : reduit
4.10 Exercice 25 : somme, produit
4.11 Exercice 26 : miroir version 2
5 Arbres
5.1 Exercice 27
5.2 Exercice 28
5.3 Exercice 29
5.4 Exercice 30

6 Parcours d'arbres binaires
6.1 Exercice 31
6.2 Exercice 32 : Parcours naifs (complexité quadratique)
6.3 Exercice 33 : Parcours linéaires
6.4 Exercice 34 : parcours en largeur et en profondeur
6.5 Exercice 35 et fin
6.5.1 Reconstruction depuis le parcours prefixe
6.5.2 Reconstruction depuis le parcours en largeur
7 Conclusion

2 TP 2 - Programmation pour la préparation à l'agrégation maths option info

- En Python, version 3.

```
In [4]: from sys import version
        print(version)
```

```
3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170118]
```

---

3 Listes

Ces exercices sont un peu foireux : les "*listes*" en Python **ne sont pas des listes simplement chaînées** !

3.1 Exercice 1 : `taille`

```
In [12]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def taille(liste : List[_a]) -> int:
             longueur = 0
             for _ in liste:
                 longueur += 1
             return longueur

         taille([])
         taille([1, 2, 3])
```

```
Out[12]: 0
```

```
Out[12]: 3
```

```
In [7]: len([])
        len([1, 2, 3])

Out[7]: 0

Out[7]: 3
```

3.2 Exercice 2 : concat

```
In [16]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def concatene(liste1 : List[_a], liste2 : List[_a]) -> List[_a]:
             # return liste1 + liste2  # easy solution
             liste = []
             for i in liste1:
                 liste.append(i)
             for i in liste2:
                 liste.append(i)
             return liste

In [19]: concatene([1, 2], [3, 4])
         [1, 2] + [3, 4]

Out[19]: [1, 2, 3, 4]

Out[19]: [1, 2, 3, 4]
```

Mais attention le typage est toujours optionnel en Python :

```
In [20]: concatene([1, 2], ["pas", "entier", "?"])

Out[20]: [1, 2, 'pas', 'entier', '?']
```

3.3 Exercice 3 : appartient

```
In [21]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def appartient(x : _a, liste : List[_a]) -> bool:
             for y in liste:
                 if x == y:
                     return True  # on stoppe avant la fin
             return False

         appartient(1, [])
         appartient(1, [1])
         appartient(1, [1, 2, 3])
         appartient(4, [1, 2, 3])
```

```
Out[21]: False

Out[21]: True

Out[21]: True

Out[21]: False
```

```
In [22]: 1 in []
         1 in [1]
         1 in [1, 2, 3]
         4 in [1, 2, 3]
```

```
Out[22]: False

Out[22]: True

Out[22]: True

Out[22]: False
```

Notre implémentation est évidemment plus lente que le test `x in liste` de la librarie standard... Mais pas tant :

```
In [23]: %timeit appartient(1000, list(range(10000)))
         %timeit 1000 in list(range(10000))
```

```
173 ţs ś 4.68 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
166 ţs ś 6.56 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

3.4 Exercice 4 : `miroir`

```
In [26]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def miroir(liste : List[_a]) -> List[_a]:
             # return liste[::-1]  # version facile
             liste2 = []
             for x in liste:
                 liste2.insert(0, x)
             return liste2
```

```
In [28]: miroir([2, 3, 5, 7, 11])
         [2, 3, 5, 7, 11][::-1]
```

```
Out[28]: [11, 7, 5, 3, 2]

Out[28]: [11, 7, 5, 3, 2]
```

```
In [29]: %timeit miroir([2, 3, 5, 7, 11])
         %timeit [2, 3, 5, 7, 11][::-1]
```

```
968 ns ś 67.4 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
216 ns ś 8.77 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

3.5 Exercice 5 : `alterne`

La sémantique n'était pas très claire, mais on peut imaginer quelque chose comme ça :

```
In [30]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def alterne(liste1 : List[_a], liste2 : List[_a]) -> List[_a]:
             liste3 = []
             i, j = 0, 0
             n, m = len(liste1), len(liste2)
             while i < n and j < m:  # encore deux
                 liste3.append(liste1[i])
                 i += 1
                 liste3.append(liste2[j])
                 j += 1
             while i < n:  # si n > m
                 liste3.append(liste1[i])
                 i += 1
             while j < m:  # ou si n < m
                 liste3.append(liste2[j])
                 j += 1
             return liste3

In [31]: alterne([3, 5], [2, 4, 6])
         alterne([1, 3, 5], [2, 4, 6])
         alterne([1, 3, 5], [4, 6])

Out[31]: [3, 2, 5, 4, 6]

Out[31]: [1, 2, 3, 4, 5, 6]

Out[31]: [1, 4, 3, 6, 5]
```

La complexité est linéaire en $\mathcal{O}(\max(|\text{liste 1}|, |\text{liste 2}|)$.

3.6 Exercice 6 : `nb_occurrences`

```
In [32]: from typing import TypeVar, List
         _a = TypeVar('alpha')

         def nb_occurrences(x : _a, liste : List[_a]) -> int:
             nb = 0
             for y in liste:
                 if x == y:
                     nb += 1
             return nb

         nb_occurrences(0, [1, 2, 3, 4])
```

```
nb_occurrences(2, [1, 2, 3, 4])
nb_occurrences(2, [1, 2, 2, 3, 2, 4])
nb_occurrences(5, [1, 2, 3, 4])
```

```
Out[32]: 0
```

```
Out[32]: 1
```

```
Out[32]: 3
```

```
Out[32]: 0
```

3.7 Exercice 7 : pairs

C'est un filtrage :

```
In [33]: filter?
```

```
In [36]: from typing import List

         def pairs(liste : List[int]) -> List[int]:
             # return list(filter(lambda x : x % 2 == 0, liste))
             return [x for x in liste if x % 2 == 0]
```

```
In [37]: pairs([1, 2, 3, 4, 5, 6])
         pairs([1, 2, 3, 4, 5, 6, 7, 100000])
         pairs([1, 2, 3, 4, 5, 6, 7, 100000000000])
         pairs([1, 2, 3, 4, 5, 6, 7, 1000000000000000000])
```

```
Out[37]: [2, 4, 6]
```

```
Out[37]: [2, 4, 6, 100000]
```

```
Out[37]: [2, 4, 6, 100000000000]
```

```
Out[37]: [2, 4, 6, 1000000000000000000]
```

3.8 Exercice 8 : range

```
In [84]: from typing import List

         def myrange(n : int) -> List[int]:
             liste = []
             i = 1
             while i <= n:
                 liste.append(i)
                 i += 1
             return liste
```

```
In [85]: myrange(4)
```

```
Out[85]: [1, 2, 3, 4]
```

```
In [47]: from typing import List

         def intervale(a : int, b : int=None) -> List[int]:
             if b == None:
                 a, b = 1, a
             liste = []
             i = a
             while i <= b:
                 liste.append(i)
                 i += 1
             return liste
```

```
In [48]: intervale(10)
         intervale(1, 4)
```

```
Out[48]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Out[48]: [1, 2, 3, 4]
```

3.9 Exercice 9 : `premiers`

Plusieurs possibilités. Un filtre d'Erathosthène marche bien, ou une filtration. Je ne vais pas utiliser de tableaux donc on est un peu réduit d'utiliser une filtration (filtrage ? *pattern matching*)

```
In [77]: def racine(n : int) -> int:
             i = 1
             for i in range(n + 1):
                 if i*i > n:
                     return i - 1
             return i

         racine(1)
         racine(5)
         racine(102)
         racine(120031)
```

```
Out[77]: 1
```

```
Out[77]: 2
```

```
Out[77]: 10
```

```
Out[77]: 346
```

```
In [78]: from typing import List

         def intervale2(a : int, b : int, pas : int=1) -> List[int]:
             assert pas > 0
```

```
            liste = []
            i = a
            while i <= b:
                liste.append(i)
                i += pas
            return liste
```

```
In [79]: intervale2(2, 12, 1)
         intervale2(2, 12, 3)
```

```
Out[79]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
Out[79]: [2, 5, 8, 11]
```

Une version purement fonctionnelle est moins facile qu'une version impérative avec une référence booléenne.

```
In [80]: def estDivisible(n : int, k : int) -> bool:
             return (n % k) == 0
```

```
In [81]: estDivisible(10, 2)
         estDivisible(10, 3)
         estDivisible(10, 4)
         estDivisible(10, 5)
```

```
Out[81]: True
```

```
Out[81]: False
```

```
Out[81]: False
```

```
Out[81]: True
```

```
In [107]: def estPremier(n : int) -> bool:
              return (n == 2) or (n == 3) or not any(map(lambda k: estDivisible(n, k), interval
```

```
In [108]: for n in range(2, 20):
              print(n, list(map(lambda k: estDivisible(n, k), intervale2(2, racine(n), 1))))
```

```
2 []
3 []
4 [True]
5 [False]
6 [True]
7 [False]
8 [True]
9 [False, True]
10 [True, False]
11 [False, False]
12 [True, True]
```

```
13 [False, False]
14 [True, False]
15 [False, True]
16 [True, False, True]
17 [False, False, False]
18 [True, True, False]
19 [False, False, False]
```

```
In [109]: from typing import List

          def premiers(n : int) -> List[int]:
              return [p for p in intervale2(2, n, 1) if estPremier(p)]
```

```
In [110]: premiers(10)

Out[110]: [2, 3, 5, 7]
```

```
In [111]: premiers(100)

Out[111]: [2,
           3,
           5,
           7,
           11,
           13,
           17,
           19,
           23,
           29,
           31,
           37,
           41,
           43,
           47,
           53,
           59,
           61,
           67,
           71,
           73,
           79,
           83,
           89,
           97]
```

4 Quelques tris par comparaison

On fera les tris en ordre croissant.

```
In [112]: test = [3, 1, 8, 4, 5, 6, 1, 2]
```

4.1 Exercice 10 : Tri insertion

```
In [113]: from typing import TypeVar, List
          _a = TypeVar('alpha')

          def insere(x : _a, liste : List[_a]) -> List[_a]:
              if len(liste) == 0:
                  return [x]
              else:
                  t, q = liste[0], liste[1:]
                  if x <= t:
                      return [x] + liste
                  else:
                      return [t] + insere(x, q)
```

```
In [114]: def tri_insertion(liste : List[_a]) -> List[_a]:
              if len(liste) == 0:
                  return []
              else:
                  t, q = liste[0], liste[1:]
                  return insere(t, tri_insertion(q))
```

```
In [115]: tri_insertion(test)
```

```
Out[115]: [1, 1, 2, 3, 4, 5, 6, 8]
```

Complexité en temps $\mathcal{O}(n^2)$.

4.2 Exercice 11 : Tri insertion générique

```
In [121]: from typing import TypeVar, List, Callable
          _a = TypeVar('alpha')

          def insere2(ordre : Callable[[_a, _a], bool], x : _a, liste : List[_a]) -> List[_a]:
              if len(liste) == 0:
                  return [x]
              else:
                  t, q = liste[0], liste[1:]
                  if ordre(x, t):
                      return [x] + liste
                  else:
                      return [t] + insere2(ordre, x, q)
```

```
In [122]: def tri_insertion2(ordre : Callable[[_a, _a], bool], liste : List[_a]) -> List[_a]:
              if len(liste) == 0:
                  return []
              else:
                  t, q = liste[0], liste[1:]
                  return insere2(ordre, t, tri_insertion2(ordre, q))

In [123]: ordre_croissant = lambda x, y: x <= y

In [124]: tri_insertion2(ordre_croissant, test)

Out[124]: [1, 1, 2, 3, 4, 5, 6, 8]

In [125]: ordre_decroissant = lambda x, y: x >= y

In [126]: tri_insertion2(ordre_decroissant, test)

Out[126]: [8, 6, 5, 4, 3, 2, 1, 1]
```

4.3 Exercice 12 : Tri selection

```
In [127]: from typing import TypeVar, List, Tuple
          _a = TypeVar('alpha')

          def selectionne_min(liste : List[_a]) -> Tuple[_a, List[_a]]:
              if len(liste) == 0:
                  raise ValueError("Selectionne_min sur liste vide")
              else:
                  def cherche_min(mini : _a, autres : List[_a], reste : List[_a]) -> Tuple[_a,
                      if len(reste) == 0:
                          return (mini, autres)
                      else:
                          t, q = reste[0], reste[1:]
                          if t < mini:
                              return cherche_min(t, [mini] + autres, q)
                          else:
                              return cherche_min(mini, [t] + autres, q)
                  t, q = liste[0], liste[1:]
                  return cherche_min(t, [], q)

In [129]: test
          selectionne_min(test)

Out[129]: [3, 1, 8, 4, 5, 6, 1, 2]

Out[129]: (1, [2, 1, 6, 5, 4, 8, 3])
```

(On voit que la liste autre a été inversée)

```
In [130]: def tri_selection(liste : List[_a]) -> List[_a]:
              if len(liste) == 0:
                  return []
              else:
                  mini, autres = selectionne_min(liste)
                  return [mini] + tri_selection(autres)

In [131]: tri_selection(test)

Out[131]: [1, 1, 2, 3, 4, 5, 6, 8]
```

Complexité en temps : $\mathcal{O}(n^2)$.

4.4 Exercices 13, 14, 15 : Tri fusion

```
In [132]: from typing import TypeVar, List, Tuple
          _a = TypeVar('alpha')

          def separe(liste : List[_a]) -> Tuple[List[_a], List[_a]]:
              if len(liste) == 0:
                  return ([], [])
              elif len(liste) == 1:
                  return ([liste[0]], [])
              else:
                  x, y, q = liste[0], liste[1], liste[2:]
                  a, b = separe(q)
                  return ([x] + a, [y] + b)

Out[132]: ([3, 8, 5, 1], [1, 4, 6, 2])

In [133]: test
          separe(test)

Out[133]: [3, 1, 8, 4, 5, 6, 1, 2]

Out[133]: ([3, 8, 5, 1], [1, 4, 6, 2])

In [134]: def fusion(liste1 : List[_a], liste2 : List[_a]) -> List[_a]:
              if (len(liste1), len(liste2)) == (0, 0):
                  return []
              elif len(liste1) == 0:
                  return liste2
              elif len(liste2) == 0:
                  return liste1
              else:  # les deux sont non vides
                  x, a = liste1[0], liste1[1:]
                  y, b = liste2[0], liste2[1:]
                  if x <= y:
                      return [x] + fusion(a, [y] + b)
```

```
                else:
                    return [y] + fusion([x] + a, b)


        fusion([1, 3, 7], [2, 3, 8])
```

```
Out[134]: [1, 2, 3, 3, 7, 8]
```

```
In [136]: def tri_fusion(liste : List[_a]) -> List[_a]:
              if len(liste) <= 1:
                  return liste
              else:
                  a, b = separe(liste)
                  return fusion(tri_fusion(a), tri_fusion(b))
```

```
In [137]: tri_fusion(test)
```

```
Out[137]: [1, 1, 2, 3, 4, 5, 6, 8]
```

Complexité en temps $\mathcal{O}(n \log n)$.

4.5 Comparaisons

```
In [138]: %timeit tri_insertion(test)
          %timeit tri_selection(test)
          %timeit tri_fusion(test)
```

```
11.8 ţs ś 463 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
52.9 ţs ś 5.31 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
32.1 ţs ś 1.14 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

```
In [152]: from sys import setrecursionlimit
          setrecursionlimit(100000)
          # nécessaire pour tester les différentes fonctions récursives sur de grosses listes
```

```
In [153]: import random

          def test_random(n : int) -> List[int]:
              return [random.randint(-1000, 1000) for _ in range(n)]

          for n in [10, 100, 1000]:
              print("\nFor n =", n)
              for tri in [tri_insertion, tri_selection, tri_fusion]:
                  print("    and tri = {}".format(tri.__name__))
                  %timeit tri(test_random(n))
```

```
For n = 10
```

```
    and tri = tri_insertion
33.2 µs ś 4.03 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
    and tri = tri_selection
85.8 µs ś 4.57 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
    and tri = tri_fusion
59.6 µs ś 5.57 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)

For n = 100
    and tri = tri_insertion
2.01 ms ś 93.3 µs per loop (mean ś std. dev. of 7 runs, 100 loops each)
    and tri = tri_selection
4.32 ms ś 123 µs per loop (mean ś std. dev. of 7 runs, 100 loops each)
    and tri = tri_fusion
1.12 ms ś 10.6 µs per loop (mean ś std. dev. of 7 runs, 1000 loops each)

For n = 1000
    and tri = tri_insertion
758 ms ś 18.3 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
    and tri = tri_selection
1.54 s ś 43.4 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
    and tri = tri_fusion
26.1 ms ś 631 µs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

- C'est assez pour vérifier que le tri fusion est **bien plus efficace** que les autres.
- On voit aussi que les tris par insertion et sélection sont pire que linéaires,
- Mais que le tri par fusion est presque linéaire (pour n petits, $n \log n$ est presque linéaire).

---

5 Listes : l'ordre supérieur

Je ne corrige pas les questions qui étaient traitées dans le TP1.

5.1 Exercice 16 : `applique`

```
In [155]: from typing import TypeVar, List, Callable
          _a, _b = TypeVar('_a'), TypeVar('_b')

          def applique(f : Callable[[_a], _b], liste : List[_a]) -> List[_b]:
              # Triche :
              return list(map(f, liste))
              # 1ère approche :
              return [f(x) for x in liste]
              # 2ème approche :
              fliste = []
              for x in liste:
                  fliste.append(f(x))
```

```
            return fliste
            # 3ème approche
            n = len(liste)
            if n == 0: return []
            fliste = [liste[0] for _ in range(n)]
            for i in range(n):
                fliste[i] = f(liste[i])
            return fliste
```

5.2 Exercice 17

```
In [156]: def premiers_carres_parfaits(n : int) -> List[int]:
              return applique(lambda x : x * x, list(range(1, n + 1)))

In [157]: premiers_carres_parfaits(12)

Out[157]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

5.3 Exercice 18 : `itere`

```
In [158]: from typing import TypeVar, List, Callable
          _a = TypeVar('_a')

          def itere(f : Callable[[_a], None], liste : List[_a]) -> None:
              for x in liste:
                  f(x)
```

5.4 Exercice 19

```
In [161]: print_int = lambda i: print("{}".format(i))

In [163]: def affiche_liste_entiers(liste : List[int]) -> None:
              print("Debut")
              itere(print_int, liste)
              print("Fin")

          affiche_liste_entiers([1, 2, 4, 5, 12011993])
```

```
Debut
1
2
4
5
12011993
Fin
```

5.5 Exercice 20 : qqsoit et ilexiste

```
In [164]: from typing import TypeVar, List, Callable
          _a = TypeVar('_a')

          # Comme all(map(f, liste))
          def qqsoit(f : Callable[[_a], bool], liste : List[_a]) -> bool:
              for x in liste:
                  if not f(x): return False   # arret preliminaire
              return True

In [176]: # Comme any(map(f, liste))
          def ilexiste(f : Callable[[_a], bool], liste : List[_a]) -> bool:
              for x in liste:
                  if f(x): return True   # arret preliminaire
              return False

In [177]: qqsoit(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5])
          ilexiste(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5])

Out[177]: False

Out[177]: True

In [167]: %timeit qqsoit(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5])
          %timeit all(map(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5]))

368 ns ś 14.9 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
495 ns ś 44.2 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)


In [168]: %timeit ilexiste(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5])
          %timeit any(map(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5]))

395 ns ś 41.7 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
641 ns ś 32.2 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

5.6 Exercice 21 : appartient version 2

```
In [178]: def appartient_curri(x : _a) -> Callable[[List[_a]], bool]:
              return lambda liste: ilexiste(lambda y: x == y, liste)

          def appartient(x : _a, liste : List[_a]) -> bool:
              return ilexiste(lambda y: x == y, liste)

In [179]: def toutes_egales(x : _a, liste : List[_a]) -> bool:
              return qqsoit(lambda y: x == y, liste)
```

```
In [181]: appartient_curri(1)([1, 2, 3])

          appartient(1, [1, 2, 3])
          appartient(5, [1, 2, 3])

          toutes_egales(1, [1, 2, 3])
          toutes_egales(5, [1, 2, 3])

Out[181]: True

Out[181]: True

Out[181]: False

Out[181]: False

Out[181]: False
```

Est-ce que notre implémentation peut être plus rapide que le test `x in liste` ? Non, mais elle est aussi rapide. C'est déjà pas mal !

```
In [183]: %timeit appartient(random.randint(-10, 10), [random.randint(-1000, 1000) for _ in ra
          %timeit random.randint(-10, 10) in [random.randint(-1000, 1000) for _ in range(1000)

1.17 ms ś 25.9 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
1.1 ms ś 10.8 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

5.7 Exercice 22 : `filtre`

```
In [186]: from typing import TypeVar, List, Callable
          _a = TypeVar('_a')

          # Comme list(filter(f, liste))
          def filtre(f : Callable[[_a], bool], liste : List[_a]) -> List[_a]:
              # return [x for x in liste if f(x)]
              liste2 = []
              for x in liste:
                  if f(x):
                      liste2.append(x)
              return liste2

In [185]: filtre(lambda x: (x % 2) == 0, [1, 2, 3, 4, 5])
          filtre(lambda x: (x % 2) != 0, [1, 2, 3, 4, 5])

Out[185]: [2, 4]

Out[185]: [1, 3, 5]
```

5.8 Exercice 23

Je vous laisse trouver pour `premiers`.

```
In [189]: pairs   = lambda liste: filtre(lambda x: (x % 2) == 0, liste)
          impairs = lambda liste: filtre(lambda x: (x % 2) != 0, liste)

In [188]: pairs(list(range(10)))
          impairs(list(range(10)))

Out[188]: [0, 2, 4, 6, 8]

Out[188]: [1, 3, 5, 7, 9]
```

5.9 Exercice 24 : `reduit`

```
In [211]: from typing import TypeVar, List, Callable
          _a = TypeVar('_a')

          # Comme list(filter(f, liste))
          def reduit_rec(f : Callable[[_a, _b], _a], acc : _a, liste : List[_b]) -> _a:
              if len(liste) == 0:
                  return acc
              else:
                  h, q = liste[0], liste[1:]
                  return reduit(f, f(acc, h), q)

          # Version non récursive, bien plus efficace
          def reduit(f : Callable[[_a, _b], _a], acc : _a, liste : List[_b]) -> _a:
              acc_value = acc
              for x in liste:
                  acc_value = f(acc_value, x)
              return acc_value
```

Très pratique pour calculer des sommes, notamment.

5.10 Exercice 25 : `somme`, `produit`

```
In [212]: from operator import add
          somme_rec = lambda liste: reduit_rec(add, 0, liste)
          somme = lambda liste: reduit(add, 0, liste)

          somme_rec(list(range(10)))
          somme(list(range(10)))
          sum(list(range(10)))

Out[212]: 45

Out[212]: 45
```

```
Out[212]: 45
```

```
In [213]: %timeit somme_rec(list(range(10)))
          %timeit somme(list(range(10)))
          %timeit sum(list(range(10)))
```

```
1.59 µs ś 16.5 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
1.42 µs ś 85.8 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
644 ns ś 7.56 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

Pour de petites listes, la version récursive est aussi efficace que la version impérative. Chouette !

```
In [214]: %timeit somme_rec(list(range(1000)))
          %timeit somme(list(range(1000)))
          %timeit sum(list(range(1000)))
```

```
84.8 µs ś 1.34 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
83.1 µs ś 2.27 µs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
19.9 µs ś 1.05 µs per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

```
In [205]: from operator import mul
          produit = lambda liste: reduit(mul, 1, liste)

          produit(list(range(1, 6)))  # 5! = 120
```

```
Out[205]: 120
```

Bonus :

```
In [208]: def factorielle(n : int) -> int:
              return produit(range(1, n + 1))

          for n in range(1, 15):
              print("{:>7}! = {:>13}".format(n, factorielle(n)))
```

```
      1! =             1
      2! =             2
      3! =             6
      4! =            24
      5! =           120
      6! =           720
      7! =          5040
      8! =         40320
      9! =        362880
     10! =       3628800
     11! =      39916800
     12! =     479001600
     13! =    6227020800
     14! =   87178291200
```

5.11 Exercice 26 : `miroir` version 2

```
In [220]: miroir = lambda liste: reduit(lambda l, x : [x] + l, [], liste)

In [221]: miroir([2, 3, 5, 7, 11])

Out[221]: [11, 7, 5, 3, 2]
```

Attention en Python, les listes ne sont PAS simplement chainées, donc `lambda l, x : [x] + l` est en temps **linéaire** en $|l| = n$, pas en $\mathcal{O}(1)$ comme en Caml/OCaml pour `fun l x -> x :: l`.

---

6 Arbres

/! Les deux dernières parties sont **bien plus difficiles** en Python qu'en Caml.

6.1 Exercice 27

```
In [1]: from typing import Dict, Optional, Tuple

        # Impossible de définir un type récursivement, pas comme en Caml
        arbre_bin = Dict[str, Optional[Tuple[Dict, Dict]]]

In [37]: from pprint import pprint

In [38]: arbre_test = {'Noeud': (
                    {'Noeud': (
                        {'Noeud': (
                            {'Feuille': None},
                            {'Feuille': None}
                        )},
                        {'Feuille': None}
                    )},
                    {'Feuille': None}
                )}

In [39]: pprint(arbre_test)

{'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
                       {'Feuille': None})},
           {'Feuille': None})}
```

Avec une syntaxe améliorée, on se rapproche de très près de la syntaxe de Caml/OCaml :

```
In [40]: Feuille = {'Feuille': None}
         Noeud = lambda x, y : {'Noeud': (x, y)}
```

```
In [34]: arbre_test = Noeud(Noeud(Noeud(Feuille, Feuille), Feuille), Feuille)

In [36]: pprint(arbre_test)

{'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
                       {'Feuille': None})},
           {'Feuille': None})}
```

6.2 Exercice 28

Compte le nombre de feuilles et de sommets.

```
In [5]: def taille(a : arbre_bin) -> int:
            # Pattern matching ~= if, elif,.. sur les clés de la profondeur 1
            # du dictionnaire (une seule clé)
            if 'Feuille' in a:
                return 1
            elif 'Noeud' in a:
                x, y = a['Noeud']
                return 1 + taille(x) + taille(y)

In [6]: taille(arbre_test)  # 7

Out[6]: 7
```

6.3 Exercice 29

```
In [7]: def hauteur(a : arbre_bin) -> int:
            if 'Feuille' in a:
                return 0
            elif 'Noeud' in a:
                x, y = a['Noeud']
                return 1 + max(hauteur(x), hauteur(y))

In [8]: hauteur(arbre_test)  # 3

Out[8]: 3
```

6.4 Exercice 30

Bonus. (Écrivez une fonction testant si un arbre étiqueté par des entiers est tournoi.)

---

7 Parcours d'arbres binaires

Après quelques exercices manipulant cette structure de dictionnaire, écrire la suite n'est pas trop difficile.

7.1 Exercice 31

```
In [9]: from typing import TypeVar, Union, List

        F = TypeVar('F')
        N = TypeVar('N')

        element_parcours = Union[F, N]
        parcours = List[element_parcours]
```

7.2 Exercice 32 : Parcours naifs (complexité quadratique)

```
In [10]: def parcours_prefixe(a : arbre_bin) -> parcours:
             if 'Feuille' in a:
                 return [F]
             elif 'Noeud' in a:
                 g, d = a['Noeud']
                 return [N] + parcours_prefixe(g) + parcours_prefixe(d)

         parcours_prefixe(arbre_test)

Out[10]: [~N, ~N, ~N, ~F, ~F, ~F, ~F]

In [11]: def parcours_postfixe(a : arbre_bin) -> parcours:
             if 'Feuille' in a:
                 return [F]
             elif 'Noeud' in a:
                 g, d = a['Noeud']
                 return parcours_postfixe(g) + parcours_postfixe(d) + [N]

         parcours_postfixe(arbre_test)

Out[11]: [~F, ~F, ~N, ~F, ~N, ~F, ~N]

In [12]: def parcours_infixe(a : arbre_bin) -> parcours:
             if 'Feuille' in a:
                 return [F]
             elif 'Noeud' in a:
                 g, d = a['Noeud']
                 return parcours_infixe(g) + [N] + parcours_infixe(d)

         parcours_infixe(arbre_test)

Out[12]: [~F, ~N, ~F, ~N, ~F, ~N, ~F]
```

Pourquoi ont-ils une complexité quadratique ? La concaténation (`@`) ne se fait pas en temps constant mais linéaire dans la taille de la plus longue liste.

7.3 Exercice 33 : Parcours linéaires

On ajoute une fonction auxiliaire et un argument vus qui est une liste qui stocke les élements observés dans l'ordre du parcours

```
In [13]: def parcours_prefixe2(a : arbre_bin) -> parcours:
             def parcours(vus, b):
                 if 'Feuille' in b:
                     vus.insert(0, F)
                     return vus
                 elif 'Noeud' in b:
                     vus.insert(0, N)
                     g, d = b['Noeud']
                     return parcours(parcours(vus, g), d)
             p = parcours([], a)
             return p[::-1]

         parcours_prefixe2(arbre_test)

Out[13]: [~N, ~N, ~N, ~F, ~F, ~F, ~F]

In [14]: def parcours_postfixe2(a : arbre_bin) -> parcours:
             def parcours(vus, b):
                 if 'Feuille' in b:
                     vus.insert(0, F)
                     return vus
                 elif 'Noeud' in b:
                     g, d = b['Noeud']
                     p = parcours(parcours(vus, g), d)
                     p.insert(0, N)
                     return p
             p = parcours([], a)
             return p[::-1]

         parcours_postfixe2(arbre_test)

Out[14]: [~F, ~F, ~N, ~F, ~N, ~F, ~N]

In [15]: def parcours_infixe2(a : arbre_bin) -> parcours:
             def parcours(vus, b):
                 if 'Feuille' in b:
                     vus.insert(0, F)
                     return vus
                 elif 'Noeud' in b:
                     g, d = b['Noeud']
                     p = parcours(vus, g)
                     p.insert(0, N)
                     return parcours(p, d)
             p = parcours([], a)
```

```
            return p[::-1]

        parcours_infixe2(arbre_test)
```

```
Out[15]: [~F, ~N, ~F, ~N, ~F, ~N, ~F]
```

7.4 Exercice 34 : parcours en largeur et en profondeur

Pour utiliser une file de priorité (*priority queue*), on utilise le module collections.deque.

```
In [16]: from collections import deque

         def parcours_largeur(a : arbre_bin) -> parcours:
             file = deque()
             # fonction avec effet de bord sur la file
             def vasy() -> parcours:
                 if len(file) == 0:
                     return []
                 else:
                     b = file.pop()
                     if 'Feuille' in b:
                         # return [F] + vasy()
                         v = vasy()
                         v.insert(0, F)
                         return v
                     elif 'Noeud' in b:
                         g, d = b['Noeud']
                         file.insert(0, g)
                         file.insert(0, d)
                         # return [N] + vasy()
                         v = vasy()
                         v.insert(0, N)
                         return v
             file.insert(0, a)
             return vasy()

         parcours_largeur(arbre_test)
```

```
Out[16]: [~N, ~N, ~F, ~N, ~F, ~F, ~F]
```

En remplaçant la file par une pile (une simple `list`), on obtient le parcours en profondeur, avec la même complexité.

```
In [17]: def parcours_profondeur(a : arbre_bin) -> parcours:
             pile = []
             # fonction avec effet de bord sur la file
             def vasy() -> parcours:
                 if len(pile) == 0:
                     return []
```

```
                else:
                    b = pile.pop()
                    if 'Feuille' in b:
                        # return [F] + vasy()
                        v = vasy()
                        v.append(F)
                        return v
                    elif 'Noeud' in b:
                        g, d = b['Noeud']
                        pile.append(g)
                        pile.append(d)
                        # return [N] + vasy()
                        v = vasy()
                        v.insert(0, N)
                        return v
            pile.append(a)
            return vasy()

        parcours_profondeur(arbre_test)
```

```
Out[17]: [~N, ~N, ~N, ~F, ~F, ~F, ~F]
```

7.5 Exercice 35 et fin

7.5.1 Reconstruction depuis le parcours prefixe

```
In [18]: test_prefixe = parcours_prefixe2(arbre_test)
         test_prefixe
```

```
Out[18]: [~N, ~N, ~N, ~F, ~F, ~F, ~F]
```

L'idée de cette solution est la suivante : j'aimerais une fonction récursive qui fasse le travail; le problème c'est que si on prend un parcours prefixe, soit il commence par F et l'arbre doit être une feuille; soit il est de la forme N::q où q n'est plus un parcours prefixe mais la concaténation de DEUX parcours prefixe, on ne peut donc plus appeler la fonction sur q. On va donc écrire une fonction qui prend une liste qui contient plusieurs parcours concaténé et qui renvoie l'arbre correspondant au premier parcours et ce qui n'a pas été utilisé :

```
In [22]: from typing import Tuple

         def reconstruit_prefixe(par : parcours) -> arbre_bin:
             def reconstruit(p : parcours) -> Tuple[arbre_bin, parcours]:
                 if len(p) == 0:
                     raise ValueError("parcours invalide pour reconstruit_prefixe")
                 elif p[0] == F:
                     return (Feuille, p[1:])
                 elif p[0] == N:
                     g, q = reconstruit(p[1:])
                     d, r = reconstruit(q)
```

```
                return (Noeud(g, d), r)
            # call it
            a, p = reconstruit(par)
            if len(p) == 0:
                return a
            else:
                raise ValueError("parcours invalide pour reconstruit_prefixe")
```

```
In [23]: reconstruit_prefixe([F])
```

```
Out[23]: {'Feuille': None}
```

```
In [24]: reconstruit_prefixe(test_prefixe)
```

```
Out[24]: {'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
             {'Feuille': None})},
           {'Feuille': None})}
```

Et cet exemple va échouer :

```
In [25]: reconstruit_prefixe([N, F, F] + test_prefixe)  # échoue
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-25-1df0bb84a92a> in <module>()
----> 1 reconstruit_prefixe([N, F, F] + test_prefixe)  # échoue


<ipython-input-22-c1349393389b> in reconstruit_prefixe(par)
     16         return a
     17     else:
---> 18         raise ValueError("parcours invalide pour reconstruit_prefixe")


ValueError: parcours invalide pour reconstruit_prefixe
```

7.5.2 Reconstruction depuis le parcours en largeur

Ce n'est pas évident quand on ne connait pas. L'idée est de se servir d'une file pour stocker les arbres qu'on reconstruit peu à peu depuis les feuilles. La file permet de récupérer les bons sous-arbres quand on rencontre un noeud

```
In [27]: largeur_test = parcours_largeur(arbre_test)
         largeur_test
```

```
Out[27]: [~N, ~N, ~F, ~N, ~F, ~F, ~F]
```

```
In [41]: from collections import deque

         def reconstruit_largeur(par : parcours) -> arbre_bin:
             file = deque()
             # Fonction avec effets de bord
             def lire_element(e : element_parcours) -> None:
                 if e == F:
                     file.append(Feuille)
                 elif e == N:
                     d = file.popleft()
                     g = file.popleft()  # attention à l'ordre !
                     file.append(Noeud(g, d))
             # Applique cette fonction à chaque élement du parcours
             for e in reversed(par):
                 lire_element(e)
             if len(file) == 1:
                 return file.popleft()
             else:
                 raise ValueError("parcours invalide pour reconstruit_largeur")

In [43]: largeur_test
         reconstruit_largeur(largeur_test)
         arbre_test

Out[43]: [~N, ~N, ~F, ~N, ~F, ~F, ~F]

Out[43]: {'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
             {'Feuille': None})},
           {'Feuille': None})}

Out[43]: {'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
             {'Feuille': None})},
           {'Feuille': None})}
```

Le même algorithme (enfin presque, modulo interversion de g et d) avec une pile donne une autre version de la reconstruction du parcours prefixe.

```
In [44]: from collections import deque

         def reconstruit_prefixe2(par : parcours) -> arbre_bin:
             pile = deque()
             # Fonction avec effets de bord
             def lire_element(e : element_parcours) -> None:
                 if e == F:
                     pile.append(Feuille)
                 elif e == N:
                     g = pile.pop()
                     d = pile.pop()  # attention à l'ordre !
                     pile.append(Noeud(g, d))
```

```
            # Applique cette fonction à chaque élement du parcours
            for e in reversed(par):
                lire_element(e)
            if len(pile) == 1:
                return pile.pop()
            else:
                raise ValueError("parcours invalide pour reconstruit_prefixe2")
```

```
In [45]: prefixe_test = parcours_prefixe2(arbre_test)
         prefixe_test
```

```
Out[45]: [~N, ~N, ~N, ~F, ~F, ~F, ~F]
```

```
In [46]: reconstruit_prefixe2(prefixe_test)
         arbre_test
```

```
Out[46]: {'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
             {'Feuille': None})},
           {'Feuille': None})}
```

```
Out[46]: {'Noeud': ({'Noeud': ({'Noeud': ({'Feuille': None}, {'Feuille': None})},
             {'Feuille': None})},
           {'Feuille': None})}
```

---

8 Conclusion

Fin. À la séance prochaine.