Programmation 2 Spring, 2026

TD 10: Divide-and-Conquer Algorithms 1

1. (**TP question**) Pour le problème des personnes honnêtes/malhonnêtes sur une liste `A`, supposons que vous avez 4 personnes, parmi lesquelles une seule est malhonnête.

   Pouvez-vous trouver cette personne avec une seule question ?

   Solution. Oui !

   (a) Choisissez deux personnes distinctes, par exemple `A[0]` et `A[1]`.

   (b) Posez la question à `A[0]` :

   « Selon vous, `A[1]` est-elle honnête ? »

   en appelant `isHonest(A[0], A[1])`. ·

   Premier cas : `isHonest(A[0], A[1])` renvoie **True** (`A[0]` dit que `A[1]` est honnête).

   - Si `A[0]` est honnête, elle dit la vérité, donc `A[1]` est honnête.
   - Si `A[0]` est malhonnête, alors `A[1]` ne peut pas être malhonnête (car il n'y a qu'une seule personne malhonnête), donc `A[1]` est honnête.

   Dans les deux cas, `A[1]` est honnête. Vous avez donc trouvé une personne honnête.

   Second cas : `isHonest(A[0], A[1])` renvoie **False** (`A[0]` dit que `A[1]` est malhonnête).

   Dans ce cas, au moins l'une des personnes `A[0]` ou `A[1]` est malhonnête. On peut donc renvoyer `A[2]` ou `A[3]`.

Remarque : Cette stratégie fonctionne car la majorité est honnête (3 honnêtes, 1 malhonnête). Elle illustre un principe clé utilisé dans les algorithmes plus généraux pour ce problème (voir TP).

```
def computeHonestPerson(A, isHonest):
    n = len(A)

    if n != 4:
        return

    if isHonest(A[0], A[1]):
        return A[1]
    else:
        return A[2]
```

2. (**Revision**) Donnez la définition formelle de $O(\cdot)$.

   Donnez une définition de $\Theta(\cdot)$ qui n'utilise que la notation $O(\cdot)$.

   Solution.

   —$f(n) = O(g(n))$ s'il existe des constantes $c > 0$ et $n_0 \in \mathbb{N}$ telles que pour tout $n \geq n_0$, on ait $|f(n)| \leq c \cdot |g(n)|$.

   —$f(n) = \Theta(g(n))$ si et seulement si $f(n) = O(g(n))$ et $g(n) = O(f(n))$.

3. Pour chacune des paires suivantes, dites si $f(n) = O(g(n))$, $g(n) = O(f(n))$, ou les deux (Θ) :

   (a) $f(n) = n^2$, $g(n) = n^2 + n$

   (b) $f(n) = \log n$, $g(n) = \sqrt{n}$

   (c) $f(n) = 2^n$, $g(n) = 3^n$

Solution.

(a) $n^2 = O(n^2 + n)$ et $n^2 + n = O(n^2)$. Donc $n^2 = \Theta(n^2 + n)$.

(b) $\log n = O(\sqrt{n})$ car $\sqrt{n}$ croît plus vite que $\log n$. En revanche, $\sqrt{n}$ n'est pas $O(\log n)$. Donc $\log n = O(\sqrt{n})$ seulement.

(c) $2^n = O(3^n)$ car $2^n \leq 1 \cdot 3^n$ pour tout $n \geq 0$. Mais 3^n n'est pas $O(2^n)$ car $(3/2)^n$ tend vers l'infini. Donc $2^n = O(3^n)$ seulement.

4. Montrez que $f(n) = n \log n$ n'est pas $O(n)$.

Solution.

Supposons par l'absurde que $n \log n = O(n)$. Alors il existe $c > 0$ et n_0 tels que $n \log n \leq c \cdot n$ pour tout $n \geq n_0$, c'est-à-dire $\log n \leq c$ pour tout $n \geq n_0$. C'est impossible car $\log n$ tend vers l'infini. Donc $n \log n \neq O(n)$.

5. Ordonnez les fonctions suivantes par ordre de croissance asymptotique (du plus petit au plus grand) :

$$\log n, \quad n, \quad n \log n, \quad n^2, \quad 2^n, \quad n!, \quad \sqrt{n}, \quad \frac{n}{\log n}$$

Solution.

L'ordre croissant est :

$$\log n, \quad \sqrt{n}, \quad \frac{n}{\log n}, \quad n, \quad n \log n, \quad n^2, \quad 2^n, \quad n!$$

6. Déterminez si les affirmations suivantes sont vraies ou fausses. Justifiez brièvement.

   (a) $n^2 = O(n^3)$

   (b) $n^3 = O(n^2)$

   (c) $n \log n = \Theta(n \log(n^2))$

   (d) $2^{n+1} = O(2^n)$

   (e) $2^{2n} = O(2^n)$

Solution.

(a) **Vrai.** $n^2 \leq 1 \cdot n^3$ pour $n \geq 1$.

(b) **Faux.** $n^3/n^2 = n$ tend vers l'infini, donc n^3 n'est pas borné par un multiple constant de n^2.

(c) **Vrai.** $\log(n^2) = 2 \log n$, donc $n \log(n^2) = 2n \log n = \Theta(n \log n)$.

(d) **Vrai.** $2^{n+1} = 2 \cdot 2^n = O(2^n)$ avec $c = 2$.

(e) **Faux.** $2^{2n} = (2^2)^n = 4^n$, et $4^n/2^n = 2^n$ tend vers l'infini.

7. Trouvez une fonction $f(n)$ telle que $f(n) = O(n)$ mais $f(n) \neq \Theta(n)$.

Solution.

De nombreux exemples conviennent. Par exemple $f(n) = \log n$. On a $\log n = O(n)$ car $\log n \leq n$ pour $n \geq 1$, mais $\log n \neq \Omega(n)$ car $\log n/n$ tend vers 0. Donc $\log n = O(n)$ mais pas $\Theta(n)$.

Autres exemples : $f(n) = \sqrt{n}$, $f(n) = 5$, $f(n) = n^{0.999}$, ...

8. Soient $f(n)$ et $g(n)$ deux fonctions strictement positives. Est-il vrai que $f(n) + g(n) = \Theta(\max(f(n), g(n)))$? Justifiez.

Solution.

Oui, c'est toujours vrai. Soit $h(n) = \max(f(n), g(n))$. Alors clairement $h(n) \leq f(n) + g(n) \leq 2h(n)$. Donc avec $c_1 = 1$ et $c_2 = 2$, on a $c_1 \cdot h(n) \leq f(n) + g(n) \leq c_2 \cdot h(n)$ pour tout n. Ainsi $f(n) + g(n) = \Theta(h(n))$.

9. Je pense à un nombre compris entre 1 et 100. Vous devez le deviner. Si vous vous trompez, je vous dirai si le nombre est supérieur ou inférieur à votre estimation.

   Voici les règles:

   - si vous devinez le nombre auquel je pense en un essai, vous gagnerez 5 euros.
   - si vous le devinez en deux essais, vous gagnerez 4 euros.
   - si vous gagnez en trois essais, vous gagnerez 3 euros.
   - si vous gagnez en quatre essais, vous gagnerez 2 euros.
   - si vous gagnez en cinq essais, vous gagnerez 1 euros.
   - si vous gagnez en six essais, vous gagnerez 0 euro.
   - si vous gagnez en sept essais, **vous me donnez** 1 euros.

   Devez-vous jouer à ce jeu ?

   Solution. Pour déterminer s'il faut jouer, il faut calculer le nombre minimal d'essais nécessaires dans le pire des cas pour deviner un nombre entre 1 et 100 avec des réponses « plus grand » ou « plus petit ».

   À chaque essai, on divise l'intervalle des possibilités en deux parties égales (ou presque). Le nombre d'essais nécessaires pour garantir de trouver le nombre est le plus petit entier k tel que $2^k \geq 100$.

   Calculons :

   $$2^6 = 64 \quad \text{et} \quad 2^7 = 128.$$

   Puisque $2^6 = 64 < 100$ et $2^7 = 128 \geq 100$, il faut au pire 7 essais pour être certain de trouver le nombre, même avec la meilleure stratégie possible.

   Conséquence sur le jeu : Dans le pire des cas, il faudra 7 essais. Or les règles stipulent que :

   - 6 essais rapportent 0 euro.
   - 7 essais font **perdre 1 euro**.

   Puisque vous ne pouvez pas garantir de gagner en 6 essais ou moins (le pire cas est 7), vous risquez de perdre 1 euro.

10. (**Revision**) On vous donne une liste `A` contenant n nombres distincts, déjà triés par ordre croissant.

    Vous devez écrire deux fonctions :

    — Écrivez une fonction `binarySearch_ITERATIVE` qui prend `A` et un nombre x. Elle renvoie l'indice de x dans `A` si x est présent, et `None` sinon.

    Votre algorithme doit utiliser des boucles.

    — Écrivez une fonction `binarySearch_RECURSIVE` qui prend `A`, un nombre x, ainsi que deux indices `start` et `end` délimitant la portion de la liste à explorer. Elle renvoie l'indice de x dans `A` si x est présent, et `None` sinon.

    Votre algorithme doit utiliser la récursion.

    Les deux algorithmes doivent s'exécuter en temps $O(\log n)$, où $n = \text{len(A)}$.

    Exemples :

    - `binarySearch_ITERATIVE([2, 5, 7, 9, 12], 7)` renvoie 2.
    - `binarySearch_RECURSIVE([2, 5, 7, 9, 12], 7, 0, 4)` renvoie 2.

    Solution.

```python
def binarySearch_ITERATIVE(A, x):
    start, end = 0, len(A)-1

    while end >= start:
        mid = (start+end)//2
        if x == A[mid]:
            return mid
        if x > A[mid]:
            start = mid+1
        if x < A[mid]:
            end = mid-1
    return None

import random
A = sorted( [ random.randint(0,20) for _ in range(0,10) ] )
print(A)

x = 4
print(f"Is {x} in A: ", binarySearch_ITERATIVE(A, x))
```

```python
def binarySearch_RECURSIVE(A, start, end, x):
    if end < start:
        return None

    mid = (start+end) // 2
    if x == A[mid]:
        return mid
    if x < A[mid]:
        return binarySearch_RECURSIVE(A, start, mid-1, x)
    else:
        return binarySearch_RECURSIVE(A, mid+1, end, x)

import random
A = sorted( [ random.randint(0,20) for _ in range(0,10) ] )
print(A)

x = 4
print(f"Is {x} in A: ", binarySearch_RECURSIVE(A, 0, len(A)-1, x))
```

11. On vous donne une liste `A` de valeurs **True**/**False**, telle que toutes les valeurs **False** apparaissent *avant* toutes les valeurs **True** (c'est-à-dire, la liste est de la forme `[False, ..., False, True, ..., True]`).

    Vous devez écrire deux fonctions qui trouvent le *premier* index de **True** dans un intervalle donné :

    — Écrivez une fonction `findFirstTrue_ITERATIVE` qui prend `A`, ainsi que deux indices `start` et `end`. Elle renvoie le premier index entre `start` et `end` (inclus) ayant la valeur **True**, ou `None` s'il n'existe pas.

    Votre algorithme doit utiliser des boucles.

    — Écrivez une fonction `findFirstTrue_RECURSIVE` qui prend `A`, ainsi que deux indices `start` et `end`. Elle renvoie le premier index entre `start` et `end` (inclus) ayant la valeur **True**, ou `None` s'il n'existe pas.

    Votre algorithme doit utiliser la récursion.

    Les deux algorithmes doivent s'exécuter en temps $O(\log n)$, où $n = \text{len}(A)$.

    Exemples :

    - `findFirstTrue_ITERATIVE([False, False, False, True, True], 0, 4)` renvoie 3.
    - `findFirstTrue_RECURSIVE([False, False, False], 0, 2)` renvoie `None`.

    Solution.

```python
def findFirstTrue_ITERATIVE(A, start, end):
    while start <= end:
        mid = (start + end) // 2
        if A[mid] == True:
            if mid == start or A[mid - 1] == False:
                return mid
            else:
                end = mid - 1
        else:
            start = mid + 1
    return None

def findFirstTrue_RECURSIVE(A, start, end):
    if start > end:
        return None

    mid = (start + end) // 2

    if A[mid] == True:
        if mid == start or A[mid - 1] == False:
            return mid
        else:
            return findFirstTrue_RECURSIVE(A, start, mid - 1)
    else:
        return findFirstTrue_RECURSIVE(A, mid + 1, end)

A1 = [False, False, False, True, True]
print(findFirstTrue_ITERATIVE(A1, 0, len(A1)-1))  # 3
print(findFirstTrue_RECURSIVE(A1, 0, len(A1)-1))  # 3

A2 = [True, True, True]
print(findFirstTrue_ITERATIVE(A2, 0, len(A2)-1))  # 0
print(findFirstTrue_RECURSIVE(A2, 0, len(A2)-1))  # 0

A3 = [False, False, False]
print(findFirstTrue_ITERATIVE(A3, 0, len(A3)-1))  # None
print(findFirstTrue_RECURSIVE(A3, 0, len(A3)-1))  # None
```

12. On vous donne une liste `A` de nombres, tels que `A[0] < A[len(A)-1]`. Notez que `A` *n'est pas trié*.

    Écrivez une fonction itérative `findIncreasing_ITERATIVE` qui prend `A` et renvoie un indice `i` tel que `A[i] < A[i+1]`.

    Votre algorithme doit prendre le temps $O(\log n)$, où `n = len(A)`.

    Exemples :

    - `findIncreasing_ITERATIVE([1, 3, 2, 4, 5])` pourrait renvoyer 0 (car `A[0] = 1 < 3 = A[1]`), ou 2 (car `A[2] = 2 < 4 = A[3]`), ou 3 (car `A[3] = 4 < 5 = A[4]`). N'importe lequel est accepté.
    - `findIncreasing_ITERATIVE([5, 4, 3, 2, 1, 6])` renvoie 4 (car `A[4] = 1 < 6 = A[5]`). Notez que `A[0] = 5 < 6 = A[-1]` est bien vérifié.
    - `findIncreasing_ITERATIVE([10, 8, 6, 4, 2, 12])` renvoie 4 (car `A[4] = 2 < 12 = A[5]`).

    Indice : Utilisez une recherche dichotomique (binaire). Observez que :

    - Si `A[mid] < A[mid+1]`, alors `mid` est une solution.
    - Sinon, une paire croissante doit exister soit à gauche, soit à droite. Utilisez cette propriété pour réduire l'intervalle de recherche.

    Solution.

```python
def findIncreasing_ITERATIVE(A):
    start = 0
    end = len(A)-1

    while (end-start+1) > 2:
        mid = (start + end) // 2

        if A[mid] < A[end]:
            start = mid
        else:
            end = mid

    return start

print(findIncreasing_ITERATIVE([60, 50, 40, 23, 28, 70]))
```

13. On vous donne une liste `A` de nombres, tels que `A[0] < A[len(A)-1]`. Notez que `A` *n'est pas triée*.

    Écrivez une fonction récursive `findIncreasing_RECURSIVE` qui prend `A`, ainsi que deux indices `start` et `end` (inclus), et renvoie un indice `i` (avec `start` $\leq$ `i` $<$ `end`) tel que `A[i] < A[i+1]`.

    Votre algorithme doit s'exécuter en temps $O(\log n)$, où $n = \text{len(A)}$.

    Exemples :

    - `findIncreasing_RECURSIVE([10, 8, 6, 4, 2, 12], 0, 5)` renvoie 4 (car `A[4] = 2 < 12 = A[5]`).
    - `findIncreasing_RECURSIVE([3, 1, 2, 5, 4], 0, 4)` pourrait renvoyer 1 (car `A[1] = 1 < 2 = A[2]`) ou 2 (car `A[2] = 2 < 5 = A[3]`).
    - `findIncreasing_RECURSIVE([60, 50, 40, 23, 28, 70], 0, 5)` renvoie 3.

    Solution.

```python
def findIncreasing_RECURSIVE(A, start, end):
    if (end-start+1) == 2:
        return start

    mid = (start + end) // 2

    if A[mid] < A[end]:
        return findIncreasing_RECURSIVE(A, mid, end)
    else:
        return findIncreasing_RECURSIVE(A, start, mid)

A1 = [60, 50, 40, 23, 28, 70]
print(findIncreasing_RECURSIVE(A1, 0, len(A1)-1))

A2 = [1, 3, 2, 4, 5]
print(findIncreasing_RECURSIVE(A2, 0, len(A2)-1))
```

14. Écrivez une fonction itérative `findMax_ITERATIVE` qui prend une liste `A` des entiers distinctes et renvoie le plus grand entier dans `A`.

    La liste `A` a une forme particulière :

    > dans l'ordre des indices, les nombres de `A` **augmentent strictement** jusqu'à un certain point, atteignent un **maximum unique**, puis **diminuent strictement**.

    Autrement dit, il existe un indice `p` (le pic) tel que :

    - `A[0] < A[1] < ... < A[p]` (partie croissante),
    - `A[p] > A[p+1] > ... > A[n-1]` (partie décroissante).

    Votre algorithme doit s'exécuter en temps $O(\log n)$, où `n = len(A)`.

    Exemples :

    - `findMax_ITERATIVE([1, 6, 9, 8, 2])` renvoie 9.
    - `findMax_ITERATIVE([3, 5, 7, 10, 6, 4])` renvoie 10.
    - `findMax_ITERATIVE([2, 4, 6, 8, 10])` renvoie 10 (liste strictement croissante, le maximum est à la fin).
    - `findMax_ITERATIVE([10, 8, 6, 4, 2])` renvoie 10 (liste strictement décroissante, le maximum est au début).
    - `findMax_ITERATIVE([5])` renvoie 5 (liste d'un seul élément).

    Solution.

```python
def findMax_ITERATIVE(A):
    start, end = 0, len(A) - 1

    while start < end:
        mid = (start + end) // 2

        if A[mid] < A[mid + 1]:
            start = mid + 1
        else:
            end = mid

    return A[start]

import random
n = 10
A=(sorted([ random.randint(0,100) for _ in range(0, n//2) ]) +
   sorted([ random.randint(0,100) for _ in range(0, n//2) ],reverse=True))

print("A: ", A)
print("maximum: ", findMax_ITERATIVE(A))
```

★ Puzzle ★

Le Défi des Échançons Royaux

Un roi maléfique possède n bouteilles de vin.
Un espion insaisissable en a empoisonné ***exactement une****.*

†

Le poison est d'une efficacité redoutable : une seule goutte diluée au milliardième suffit à tuer. Mais il lui faut un mois entier pour faire effet.

†

Le roi a à son service des testeurs (des goûteurs) chargés de vérifier si sa nourriture ou son vin est empoisonné.

♦ VOTRE MISSION ♦

Le roi doit décider, avant le début du test, quel testeur boit quelles bouteilles. Une fois le plan établi, les testeurs boivent. Le roi attend un mois. Au bout de ce mois, il observe quels testeurs sont morts et lesquels sont encore vivants. À partir de ce seul résultat—un motif de morts et de survivants—le roi doit pouvoir calculer avec certitude quelle bouteille était empoisonnée.

Concevez un algorithme permettant d'identifier avec certitude la bouteille empoisonnée en un seul mois, en utilisant seulement $O(\log n)$ testeurs de goût. Chaque testeur peut boire un mélange de plusieurs bouteilles. *Vous pouvez supposer que n est une puissance de 2.*

⋆ EXEMPLE ⋆

Une stratégie naive utilise n testeurs :

> le i-ième testeur goûte uniquement la i-ième bouteille.

Un mois plus tard, le testeur décédé désigne la bouteille empoisonnée.

Votre défi : réduire le nombre de testeurs à $O(\log n)$.

Solution. L'astuce consiste à faire boire tous les testeurs **au même moment**, puis à attendre un mois pour observer les résultats. On utilise $2\log_2 n$ testeurs, organisés en $\log_2 n$ paires, chaque paire correspondant à un niveau de division.

On numérote les bouteilles de 0 à $n-1$. On forme $\log_2 n$ niveaux (ou "couches"). Pour chaque niveau, on dispose de deux testeurs :

- Un testeur "gauche" qui boit toutes les bouteilles dont le numéro, dans ce niveau, se trouve dans la moitié gauche.
- Un testeur "droite" qui boit toutes les bouteilles dont le numéro, dans ce niveau, se trouve dans la moitié droite.

Tous les testeurs de tous les niveaux boivent **en même temps** au début de l'expérience. Un mois plus tard, on observe quels testeurs sont morts.

Comment reconstruire le numéro de la bouteille empoisonnée ?

Pour chaque niveau, l'un des deux testeurs meurt (celui qui a bu la bouteille empoisonnée), l'autre reste vivant. La combinaison des résultats sur tous les niveaux permet de déterminer exactement la bouteille.

Exemple avec $n = 8$ bouteilles (numérotées de 0 à 7) :

On utilise $\log_2 8 = 3$ niveaux, donc $2 \times 3 = 6$ testeurs, organisés en trois paires :

- **Niveau 1 (premier découpage) :** On sépare les bouteilles en deux groupes de 4 : la moitié gauche $\{0,1,2,3\}$ et la moitié droite $\{4,5,6,7\}$.
  - Testeur G_1 boit $\{0,1,2,3\}$.
  - Testeur D_1 boit $\{4,5,6,7\}$.
- **Niveau 2 (deuxième découpage) :** On regarde chaque moitié séparément. Pour le groupe $\{0,1,2,3\}$, on le divise en $\{0,1\}$ (gauche) et $\{2,3\}$ (droite). Pour le groupe $\{4,5,6,7\}$, on le divise en $\{4,5\}$ (gauche) et $\{6,7\}$ (droite).
  - Testeur G_2 boit les premiers éléments de chaque bloc : $\{0,1,4,5\}$.
  - Testeur D_2 boit les deuxièmes éléments de chaque bloc : $\{2,3,6,7\}$.
- **Niveau 3 (troisième découpage) :** On affine encore. Pour chaque bloc de 2, on le divise en deux bouteilles individuelles.
  - Testeur G_3 boit la première bouteille de chaque bloc de 2 : $\{0,2,4,6\}$.
  - Testeur D_3 boit la deuxième bouteille de chaque bloc de 2 : $\{1,3,5,7\}$.

Interprétation des résultats :

Supposons que la bouteille empoisonnée soit la numéro 5. Un mois plus tard, on observe :

- Au niveau 1 : 5 est dans la moitié droite $\{4,5,6,7\}$ donc D_1 meurt, G_1 vit.
- Au niveau 2 : 5 est dans la moitié gauche du bloc $\{4,5\}$ (car 5 est dans $\{4,5\}$) donc G_2 meurt, D_2 vit.
- Au niveau 3 : 5 est dans la deuxième bouteille du bloc $\{4,5\}$ donc D_3 meurt, G_3 vit.

Ainsi, les testeurs morts sont D_1, G_2, D_3. En associant à chaque niveau le choix "gauche" ou "droite", on obtient une séquence : droite - gauche - droite, qui correspond au chemin menant à la bouteille 5.

Solution 1.

```
poison = 11

def afterOneMonth(A):
    OUT = []
    for B in A:
        OUT.append( (B, poison in B) )
    return OUT

def assignTesters(n):
    A = []
    s = 1
    while s < n:
        P1 = []
        P2 = []
        i = 0
        while i < n:
            P1 = P1 + list(range(i,i+s))
            P2 = P2 + list(range(i+s,i+2*s))
            i = i + 2*s
        A.append( P1 )
        A.append( P2 )
        s = 2*s

    return A

def findPoison(B, n):
    T = set(range(0, n))
    for (S, V) in B:
        if V == False:
            T = T - set(S)
    print("The poisonous glass is: ", T)

A = assignTesters(16)
print(A)

B = afterOneMonth(A)
print(B)

findPoison(B, 16)
```

Solution 2.

```
def afterOneMonth(A, poison):
    OUT = []
    for B in A:
        OUT.append( (B, poison in B) )
    return OUT

def assignTesters(n):
    A = []
    s = 1
    while s < n:
        P = []
        i = 0
        while i < n:
            P = P + list(range(i,i+s))
            i = i + 2*s
        A.append( P )
        s = 2*s

    return A

def findPoison(B, n):
    s, e = 0, n-1
    for i in reversed(range(0,len(B))):
        print("l: ", B[i])
        if B[i][1] == True: e = (s+e)//2
        else: s = (s+e)//2 + 1
        print(s,e)

    print("The poisonous glass is: ", s, e)



n = 16
A = assignTesters(n)
for i in range(0, 4):
    poison = int(input("Enter poisonous glass:"))
    if poison > n-1: continue
    B = afterOneMonth(A, poison)
    findPoison(B, n)
```